INÍCIO

Olá, Seja Bem-Vindo ao Deep Learning Book Brasil.

Esta é uma inciativa da Data Science Academy, com o objetivo de ajudar a difundir uma das tecnologias mais revolucionárias do nosso tempo, o Deep Learning. Como existe pouco material em português, esperamos que este livro, em português e gratuito, o ajude a compreender um pouco melhor o que é a tecnologia e como ela pode ser usada na construção de aplicações de Inteligência Artificial.

Este livro vai abordar desde o básico em redes neurais artificiais, modelões Perceptron e Multilayer Perceptron, passando pelo processo de treinamento com Backpropagation, até arquiteturas avançadas de Deep Learning, como Redes Neurais Convolucionais, Redes Neurais Recorrentes, Autoencoders, Generative Adversarial Network e Deep Reinforcement Learning. Além disso, também abordaremos o uso de Deep Learning em sistemas embarcados usando RaspberryPi e Nvidia Jetson TX2 e aplicações em Visão Computacional e Processamento de Linguagem Natural. E teremos ainda álgebra linear e estatística.

Nossa primeira meta já foi alcançada e 50 capítulos já foram desenvolvidos. Nossa segunda meta é chegar a 100 capítulos, tudo 100% em português. Desta forma, esperamos contribuir para o crescimento do Deep Learning e Inteligência Artificial no Brasil.

Nos acompanhe nessa incrível jornada!

Equipe DAS

www.datascienceacademy.com.br

CAPÍTULO 1

Deep Learning e a Tempestade Perfeita

O interesse pela Aprendizagem de Máquina (Machine Learning) explodiu na última década. O mundo a nossa volta esta passando por uma transformação e vemos ua interação cada vez maior das aplicações de computador com os seres humanos. Softwares de detecção de spam, sistemas de recomendação, marcação em fotos de redes sociais, assistentes pessoais ativados por voz, carros autônomos, smartphones com reconhecimento facial e muito mais.

E o interesse por Machine Learning se mostra ainda mais evidente pelo número cada vez maior de conferências, meetups, artigos, livros, cursos, buscas no Google e profissionais e empresas procurando compreender o que é e como usar aprendizagem de máquina, embora muitos ainda confundem o que podem fazer com o que desejam fazer. Não há como ficar indiferente a esta revolução trazida pela aprendizagem de máquina e, segundo o Gartner, até 2020 todos os softwares corporativos terão funcionalidade ligada a Machine Learning.

Fundamentalmente, Machine Learning é a utilização de algoritmos para extrair informações de dados brutos e representá-los através de algum tipo de modelo matemático. Usamos então este modelo para fazer inferências a partir de outros conjuntos de dados. Existem muitos algoritmos que permitem fazer isso, mas um tipo especial vem se destacando, as redes neurais artificiais.

As redes neurais artificiais não são necessariamente novas, existem pelo menos desde a década de 1950. Mas durante várias décadas, embora a arquitetura desses modelos tivesse evoluído, ainda faltavam ingredientes que fizessem os modelos realmente funcionar. E esses ingredientes surgiram quase ao mesmo tempo. Um dele você já deve ter ouvido: Big Data. O volume de dados, gerado em variedade e velocidade cada vez maiores, permite criar modelos atingir altos níveis de precisão. Mas ainda falta um ingrediente. Faltava! Como processar grandes modelos de Machine Learning com grandes quantidades de dados? As CPUs não conseguiam dar conta do recado.

Foi quando os gamers e sua avidez por poder computacional e gráficos perfeitos, nos ajudaram a encontrar o segundo ingrediente: Programação Paralela em GPUs. As unidades de processamento gráfico, que permitem realizar operações matemáticas de forma paralela, principalmente operações com matrizes e vetores, elementos presentes em modelos de redes neurais artificiais, formaram a tempestade perfeita, que permitiu a evolução na qual nos encontramos hoje: Big Data + Processamento Paralelo + Modelos de Aprendizagem de Máquina = Inteligência Artificial.

A unidade fundamental de uma rede neural artificial é um nó (ou neurônio matemático), que por sua vez é baseado no neurônio biológico. As conexões entre esses neurônios matemáticos também foram inspiradas em cérebros biológicos, especialmente na forma como essas conexões se desenvolvem ao longo do tempo com “treinamento”. Em meados da década de 1980 e início da década de 1990, muitos avanços importantes na arquitetura das redes neurais artificiais ocorreram. No entanto, a quantidade de tempo e dados necessários para obter bons resultados retardou a adoção e, portanto, o interesse foi arrefecido, com o que ficou conhecido como AI Winter (Inverno da IA).

No início dos anos 2000, o poder computacional expandiu exponencialmente e o mercado viu uma “explosão” de técnicas computacionais que não eram possíveis antes disso. Foi quando o aprendizado profundo (Deep Learning) emergiu do crescimento computacional explosivo

dessa década como o principal mecanismo de construção de sistemas de Inteligência Artificial, ganhando muitas competições importantes de aprendizagem de máquina. O interesse por Deep Learning não para de crescer e hoje vemos o termo aprendizado profundo sendo mencionado com frequência cada vez maior e soluções comerciais surgindo a todo momento.

Este livro, gratuito e em português, é uma iniciativa da Data Science Academy para ajudar aqueles que buscam conhecimento avançado e de qualidade em nosso idioma. Desta forma, esperamos contribuir para o crescimento do Deep Learning e Inteligência Artificial no Brasil.

Nos acompanhe nessa incrível jornada!

Equipe DSA

www.datascienceacademy.com.br

CAPÍTULO 2

Uma Breve História das Redes Neurais Artificiais

Para compreender onde estamos hoje, precisamos olhar para o passado e analisar como chegamos até aqui. Vejamos então Uma Breve História das Redes Neurais Artificiais.

O cérebro humano é uma máquina altamente poderosa e complexa capaz de processar uma grande quantidade de informações em tempo mínimo. As unidades principais do cérebro são os neurônios e é por meio deles que as informações são transmitidas e processadas. As tarefas realizadas pelo cérebro intrigam os pesquisadores, como por exemplo, a capacidade de reconhecer um rosto familiar dentre uma multidão em apenas milésimos de segundo. As respostas sobre alguns enigmas do funcionamento do cérebro ainda não foram respondidas e se perpetuam até os dias de hoje. O que é conhecido sobre o funcionamento do cérebro é que o mesmo desenvolve suas regras através da experiência adquirida em situações vividas anteriormente.

Fig1 – Cérebro humano, a máquina mais fantástica que existe no Planeta Terra.

O desenvolvimento do cérebro humano ocorre principalmente nos dois primeiros anos de vida, mas se arrasta por toda a vida. Inspirando-se neste modelo, diversos pesquisadores tentaram simular o funcionamento do cérebro, principalmente o processo de aprendizagem por experiência, a fim de criar sistemas inteligentes capazes de realizar tarefas como classificação, reconhecimento de padrões, processamento de imagens, entre outras atividades. Como resultado destas pesquisas surgiu o modelo do neurônio artificial e posteriormente um sistema com vários neurônios interconectados, a chamada Rede Neural.

Em 1943, o neurofisiologista Warren McCulloch e o matemático Walter Pitts escreveram um artigo sobre como os neurônios poderiam funcionar e para isso, eles modelaram uma rede neural simples usando circuitos elétricos.

Warren McCulloch e Walter Pitts criaram um modelo computacional para redes neurais baseadas em matemática e algoritmos denominados lógica de limiar (threshold logic). Este modelo abriu o caminho para a pesquisa da rede neural dividida em duas abordagens: uma abordagem focada em processos biológicos no cérebro, enquanto a outra focada na aplicação de redes neurais à inteligência artificial.

Em 1946, Donald Hebb escreveu *The Organization os Behavior,* uma obra que apontou o fato de que os caminhos neurais são fortalecidos cada vez que são usados, um conceito fundamentalmente essencial para a maneira como os humanos aprendem. Se dois nervos dispararem ao mesmo tempo, argumentou, a conexão entre eles é melhorada.

À medida que os computadores se tornaram mais avançados na década de 1950, finalmente foi possível simular uma hipotética rede neural. O primeiro passo para isso foi feito por Nathanial Rochester dos laboratórios de pesquisa da IBM. Infelizmente para ele, a primeira tentativa de fazê-lo falhou.

No entanto, ao longo deste tempo, os defensores das "máquinas pensantes" continuaram a argumentar suas pesquisas. Em 1956, o Projeto de Pesquisa de Verão de Dartmouth sobre Inteligência Artificial proporcionou um impulso tanto à Inteligência Artificial como às Redes Neurais. Um dos resultados deste processo foi estimular a pesquisa em IA na parte de processamento neural.

Nos anos seguintes ao Projeto Dartmouth, John von Neumann sugeriu imitar funções simples de neurônios usando relés telegráficos ou tubos de vácuo. Além disso, Frank Rosenblatt, um neurobiologista, começou a trabalhar no Perceptron. Ele estava intrigado como funcionamento do olho de uma mosca. Grande parte do processamento feito por uma mosca ao decidir fugir, é feito em seus olhos. O Perceptron, que resultou dessa pesquisa, foi construído em hardware e é a mais antiga rede neural ainda em uso hoje. Um Perceptron de camada única foi útil para classificar um conjunto de entradas de valor contínuo em uma de duas classes. O Perceptron calcula uma soma ponderada das entradas, subtrai um limite e passa um dos dois valores possíveis como resultado. Infelizmente, o Perceptron é limitado e foi comprovado como tal durante os "anos desiludidos" por Marvin Minsky e o livro de Seymour Papert de 1969, Perceptrons.

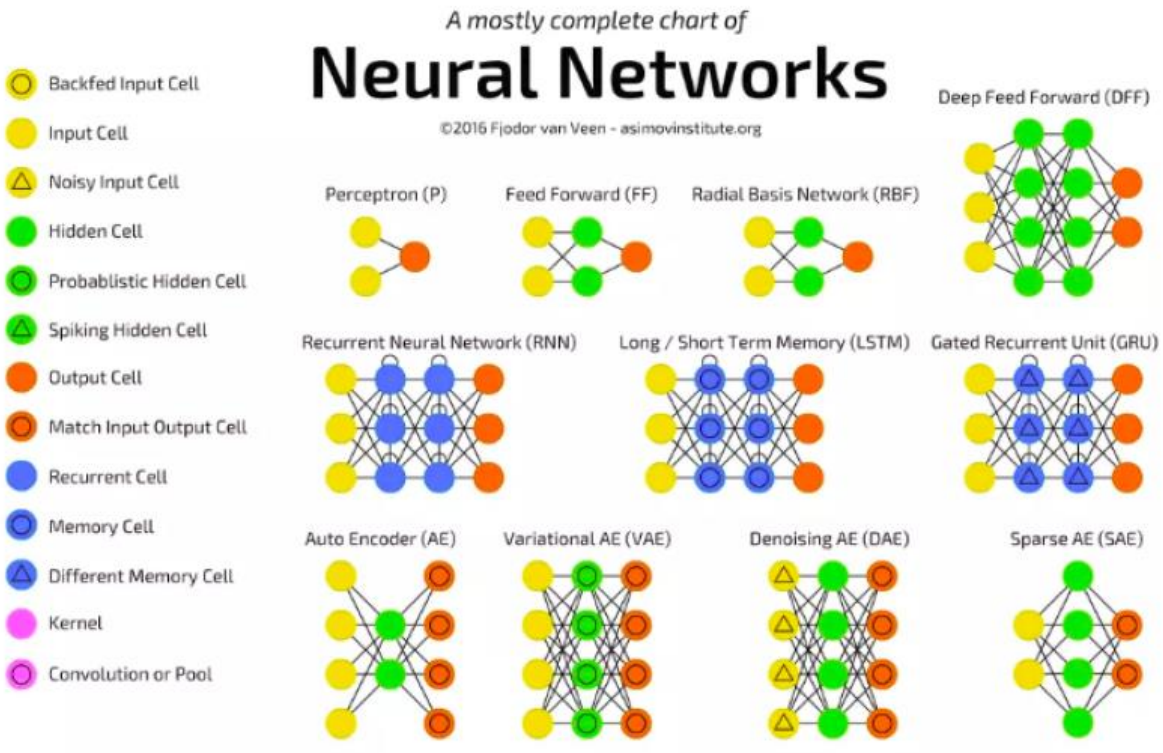


Fig2 – Algumas Arquiteturas de Redes Neurais

Em 1959, Bernard Widrow e Marcian Hoff, de Stanford, desenvolveram modelos denominados "ADALINE" e "MADALINE". Em uma exibição típica do amor de Stanford por siglas, os nomes provêm do uso de múltiplos elementos ADAptive LINear. ADALINE foi desenvolvido para reconhecer padrões binários de modo que, se ele estivesse lendo bits de transmissão de uma linha telefônica, poderia prever o próximo bit. MADALINE foi a primeira rede neural aplicada a um problema do mundo real, usando um filtro adaptativo que elimina ecos nas linhas telefônicas. Embora o sistema seja tão antigo como os sistemas de controle de tráfego aéreo, ele ainda está em uso comercial.

Infelizmente, esses sucessores anteriores levaram as pessoas a exagerar o potencial das redes neurais, particularmente à luz da limitação na eletrônica, então disponível na época. Este exagero excessivo, que decorreu do mundo acadêmico e técnico, infectou a literatura geral da época. Muitas promessas foram feitas, mas o resultado foi o desapontamento. Além disso, muitos escritores começaram a refletir sobre o efeito que teria "maquinas pensantes" no homem. A série de Asimov em robôs revelou os efeitos sobre a moral e os valores do homem quando máquinas fossem capazes de fazer todo o trabalho da humanidade. Outros escritores criaram computadores mais sinistros, como HAL do filme 2001.

Toda essa discussão sobre o efeito da Inteligência Artificial sobre a vida humana, aliada aos poucos progressos, fizeram vozes respeitadas criticar a pesquisa em redes neurais. O resultado foi a redução drástica de grande parte do financiamento em pesquisas. Esse período de crescimento atrofiado durou até 1981, sendo conhecido como o Inverno da IA (AI Winter).

Em 1982, vários eventos provocaram um renovado interesse. John Hopfield da Caltech apresentou um documento à Academia Nacional de Ciências. A abordagem de Hopfield não era simplesmente modelar cérebros, mas criar dispositivos úteis. Com clareza e análise matemática, ele mostrou como essas redes poderiam funcionar e o que poderiam fazer. No entanto, o maior recurso de Hopfield foi seu carisma. Ele era articulado e simpático e isso colaborou bastante para que ele fosse ouvido.

Em 1985, o Instituto Americano de Física começou o que se tornou uma reunião anual – Redes Neurais para Computação. Em 1987, a primeira Conferência Internacional sobre redes Neurais do nstitute of Electrical and Electronic Engineer's (IEEE) atraiu mais de 1.800 participantes.

Em 1986, com redes neurais de várias camadas nas notícias, o problema era como estender a regra Widrow-Hoff para várias camadas. Três grupos independentes de pesquisadores, dentre os quais David Rumelhart, ex-membro do departamento de psicologia de Stanford, apresentaram ideias semelhantes que agora são chamadas de redes Backpropagation porque distribuem erros de reconhecimento de padrões em toda a rede. As redes híbridas utilizavam apenas duas camadas, essas redes de Backpropagation utilizam muitas. O resultado é que as redes de Backpropagation "aprendem" de forma mais lenta, pois necessitam, possivelmente, de milhares de iterações para aprender, mas geram um resultado muito preciso.

Agora, as redes neurais são usadas em várias aplicações. A ideia fundamental por trás da natureza das redes neurais é que, se ela funcionar na natureza, deve ser capaz de funcionar em computadores. O futuro das redes neurais, no entanto, reside no desenvolvimento de hardware. As redes neurais rápidas e eficientes dependem do hardware especificado para seu eventual uso.

O diagrama abaixo mostra alguns marcos importantes na evolução e pesquisa das redes neurais artificiais. O fato, é que ainda estamos escrevendo esta história e muita evolução está ocorrendo neste momento, através do trabalho de milhares de pesquisadores e profissionais de Inteligência Artificial em todo mundo. E você, não quer ajudar a escrever esta história?

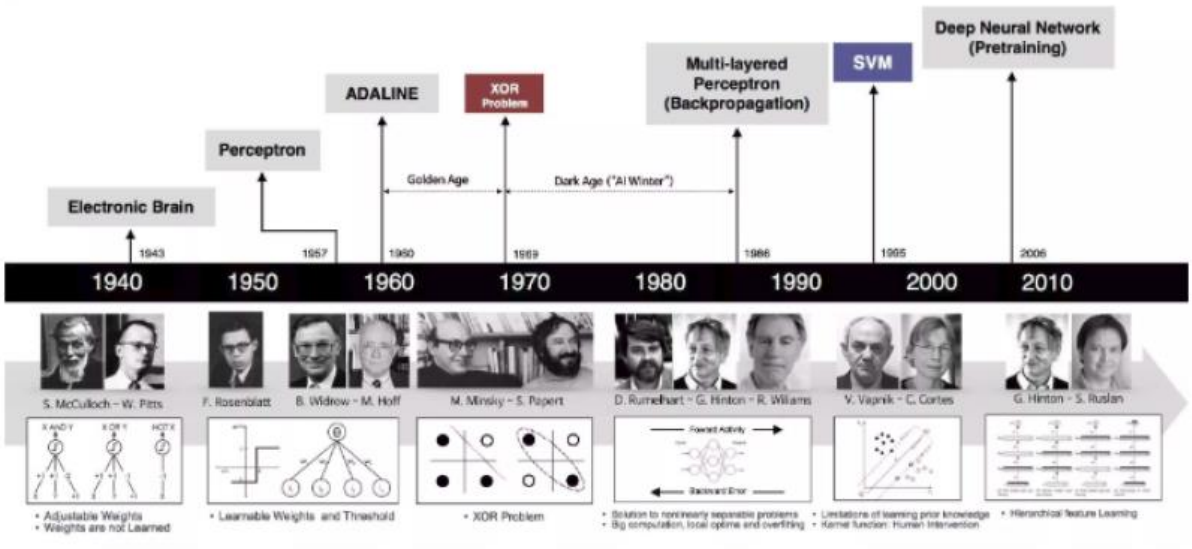


Fig3 – Marcos no desenvolvimento das redes neurais.

Podemos resumir assim os principais marcos na pesquisa e evolução das redes neurais artificiais até chagarmos ao Deep Learning:

1943: Warren McCullock e Walter Pitts criam um modelo computacional para redes neurais baseadas em matemática e algoritmos denominados lógica de limiar.

1958: Frank Rosenblatt cria o Perceptron, um algoritmo para o reconhecimento de padrões baseado em uma rede neural computacional de duas camadas usando simples adição e subtração. Ele também propôs camadas com notações matemáticas, mas isso não seria realizado até 1975.

1980: Kunihiko Fukushima propõe a Neoconitron, uma rede neural de hierarquia, multicamada, que foi utilizada para o reconhecimento de caligrafia e outros problemas de reconhecimento de padrões.

1989: Os cientistas conseguiram criar algoritmos que usavam redes neurais profundas, mas os tempos de treinamento para os sistemas foram medidos em dias, tornando-os impraticáveis para o uso no mundo real.

1992: Juyang Weng publica o Cresceptron, um método para realizar o reconhecimento de objetos 3-D automaticamente a partir de cenas desordenadas.

Meados dos anos 2000: O termo "aprendizagem profunda" começa a ganhar popularidade após um artigo de Geoffrey Hinton e Ruslan Salakhutdinov mostrar como uma rede neural de várias camadas poderia ser pré-trinada uma camada por vez.

2009: Acontece o NIPS Workshop sobre Aprendizagem Profuna para Reconhecimento de Voz e descobre-se que com um conjunto de dados suficientemente grande, as redes neurais não precisam de pré-treinamento e as taxas de erro caem significativamente.

2012: Algoritmos de reconhecimento de padrões artificiais alcançam desempenho em nível humano em determinadas tarefas. E o algoritmo de aprendizagem profunda do Google é capaz de identificar gatos.

2014: O Google compra a Startup de Inteligência Artificial chamada DeepMind, do Reino Unido, por £ 400m.

2015: Facebook coloca a tecnologia de aprendizado profundo – chamada DeepFace – em operação para marcar e identificar automaticamente usuários do Facebook em fotografias. Algoritmos executam tarefas superiores de reconhecimento facial usando redes profundas que levam em conta 120 milhões de parâmetros.

2016: O algoritmo do Google DeepMind, AlphaGo, mapeia a arte do complexo jogo de tabuleiro Go e vence o campeão mundial de Go, Lee Sedol, em um torneio altamente divulgado em Seul.

2017: Adoção em massa do Deep Learning em diversas aplicações corporativas e mobile, além do avanço em pesquisas. Todos os eventos de tecnologia ligados a data Science, IA e Big Data, apontam Deep Learning como a principal tecnologia para a criação de sistemas inteligentes.

A promessa do aprendizado profundo não é que os computadores comecem a pensar como seres humanos. Isso é como pedir uma maçã para se tornar uma laranja. Em vez disso, demonstra que, dado um conjunto de dados suficientemente grande, processadores rápidos e um algoritmo suficientemente sofisticado, os computadores podem começar a realizar tarefas que até então só podiam ser realizadas apenas por seres humanos, como reconhecer imagens e voz, criar obras de arte ou tomar decisões por si mesmo.

Os estudos sobre as redes neurais sofreram uma grande revolução a partir dos anos 80 e esta área de estudos tem se destacado, seja pelas promissoras características apresentadas pelos modelos de redes neurais propostos, seja pelas condições tecnológicas atuais de implementação que permitem desenvolver arrojadas implementações de arquiteturas neurais paralelas em hardwares dedicados, obtendo assim ótimas performances destes sistemas (bastante superiores aos sistemas convencionais). A evolução natural das redes neurais, s]ao as redes neurais profundas (ou Deep Learning). Mas isso é o que vamos discutir no próximo capítulo! Até lá.

Referências:

Christopher D. Manning. (2015). Computational Linguistics and Deep Learning Computational Linguistics, 41(4), 701–707.

F. Rosenblatt. The perceptron, a perceiving and recognizing automaton Project Para. Cornell Aeronautical Laboratory, 1957.

W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. The bulletin of mathematical biophysics, 5(4):115–133, 1943.

The organization of behavior: A neuropsychological theory. D. O. Hebb. John Wiley And Sons, Inc., New York, 1949

B. Widrow et al. Adaptive ”Adaline” neuron using chemical ”memistors”. Number Technical Report 1553-2. Stanford Electron. Labs., Stanford, CA, October 1960.

“New Navy Device Learns By Doing”, New York Times, July 8, 1958.

Perceptrons. An Introduction to Computational Geometry. MARVIN MINSKY and SEYMOUR PAPERT. M.I.T. Press, Cambridge, Mass., 1969.

Minsky, M. (1952). A neural-analogue calculator based upon a probability model of reinforcement. Harvard University Pychological Laboratories internal report.

CAPÍTULO 3

O que são Redes Neurais Artificiais Profundas ou Deep Learning?

Aprendizagem Profunda ou Deep Learning, é uma sub-área da Aprendizagem de Máquina, que emprega algoritmos para processar dados e imitar o processamento feito pelo cérebro humano. Mas O Que São Redes Neurais Artificiais Profundas ou Deep Learning? É o que veremos neste capítulo. Não se preocupe se alguns termos mais técnicos não fizerem sentido agora. Todos eles serão estudados ao longo deste livro.

Deep Learning usa camadas de neurônios matemáticos para processas dados, compreender a fala humana e reconhecer objetos visualmente. A informação é passada através de cada camada, coma saída da camada anterior fornecendo entrada para a próxima camada. A primeira camada em uma rede é chamada de camada de entrada, enquanto a última é chamada de camada de saída. Todas as camadas entre as duas são referidas como camadas ocultas. Cada camada é tipicamente um algoritmo simples e uniforme contendo um tipo de função de ativação.

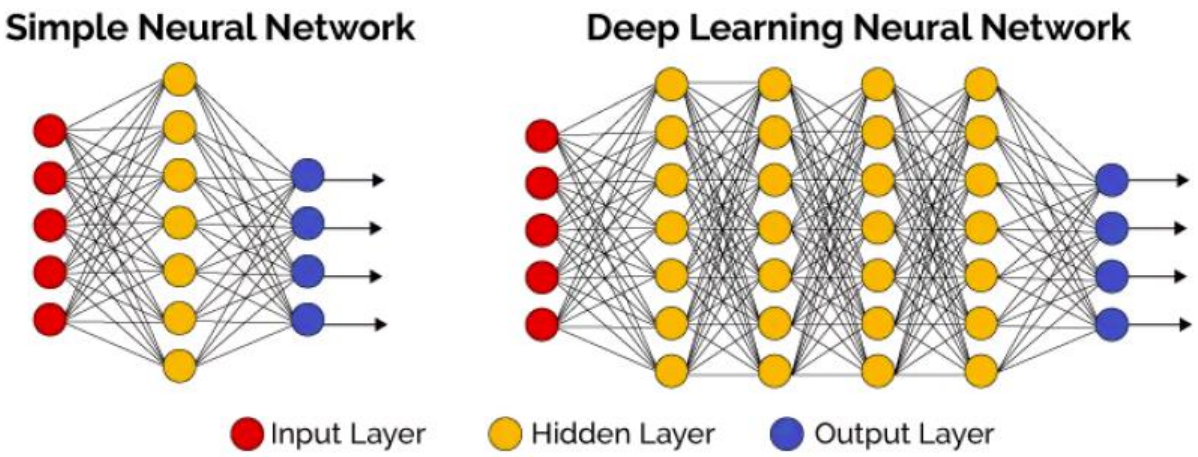


Fig4 – Rede Neural Simples e rede Neural profunda (Deep Learning)

A aprendizagem profunda é responsável por avanços recentes em visão computacional, reconhecimento de fala, processamento de linguagem natural e reconhecimento de áudio. O aprendizado profundo é baseado no conceito de redes neurais artificiais, ou sistemas computacionais que imitam a maneira como o cérebro humano funciona.

A extração de recursos é outro aspecto da Aprendizagem Profunda. A extração de recursos usa um algoritmo para construir automaticamente “recursos” significativos dos dados para fins de treinamento, aprendizado e compreensão. Normalmente, o Cientista de Dados, ou engenheiro de IA, é responsável pela extração de recursos.

O aumento rápido e o aparente domínio do aprendizado profundo sobre os métodos tradicionais de aprendizagem de máquina em uma variedade de tarefas tem sido surpreendente de testemunhar e, as vezes, difícil de explicar. Deep Learning é uma evolução das Redes Neurais, que por sua vez possuem uma história fascinante que remonta à década de 1940, cheia de altos e baixos, voltas e reviravoltas, amigos e rivais, sucessos e fracassos. Em

uma história digna de um filme dos anos 90, uma ideia que já foi uma espécie de patinho feio floresceu para tornar a bola da vez.

Consequentemente, o interesse em aprendizagem profunda tem disparado, com cobertura constante na mídia popular. A pesquisa de aprendizagem profunda agora pararece rotineiramente em revistas como Science, Nature, Nature Methods e Forbes apenas para citar algumas. O aprendizado profundo conquistou Go, aprendeu a dirigir um carro, diagnosticou câncer de pele e autismo, tornou-se um falsificador de arte e pode até alucinar imagens fotorrealistas.

Os primeiros algoritmos de aprendizagem profunda que possuíam camadas de características não-lineares podem ser rastreados até Alexey Grigoryevich Ivakhnenko (desenvolveu o Método do Grupo de Manipulação de Dados) e Valentin Grigor'evich Lapa (autor de Cybernetics and Forecasting Techniques) em 1965 (Figura 5), que usaram modelos finos mas profundos com funções de ativação polinomial os quais eles analisaram com métodos estatísticos. Em cada camada, eles selecionavam os melhores recursos através de métodos estatísticos e encaminhavam para a próxima camada. Eles não usaram Backpropagation para treinar a rede de ponta a ponta, mas utilizaram mínimos quadrados camada-por-camada, onde as camada anteriores foram independentemente instaladas em camadas posteriores (um processo lento e manual).

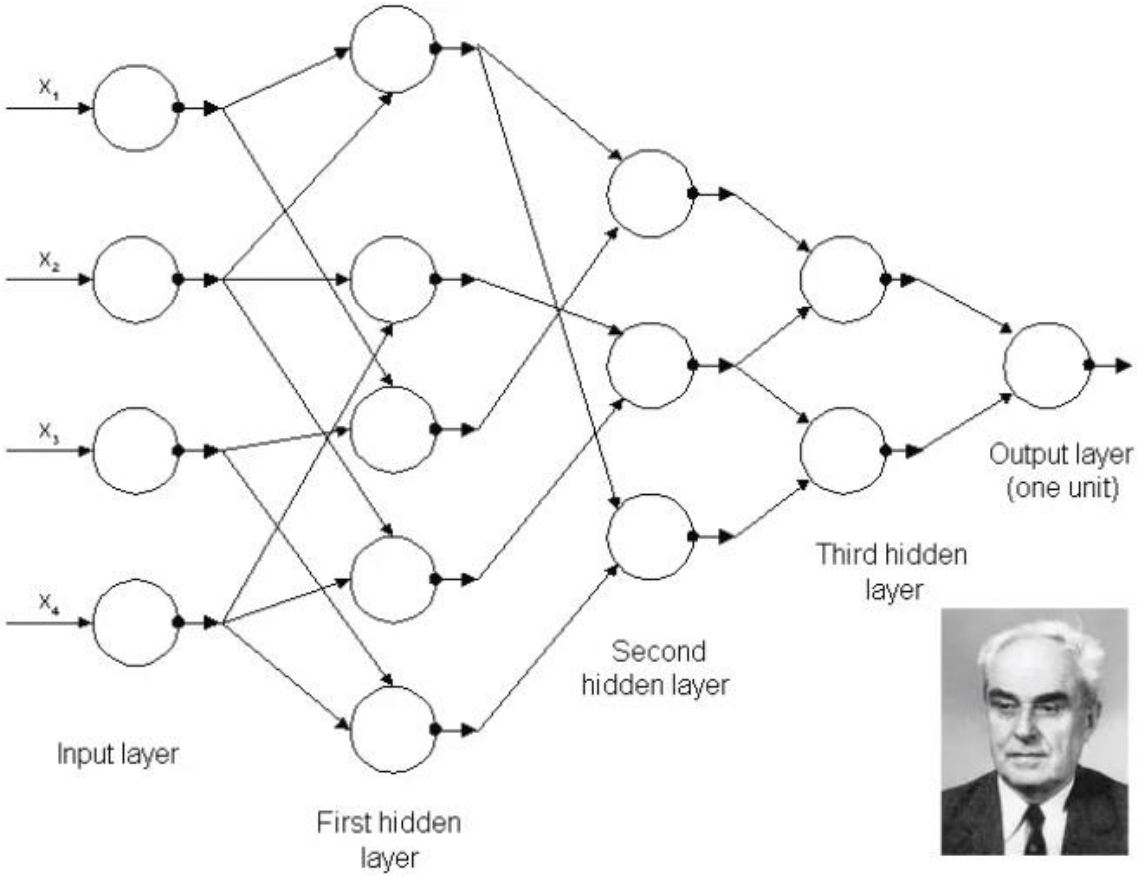


Fig5 – Arquitetura da primeira rede profunda conhecida treinada por Alexey Grigorevich Ivakhnenko em 1965.

No final da década de 1970, o primeiro inverno de AI começou, resultado de promessas que não poderiam ser mantidas. O impacto desta falta de financiamento limitou a pesquisa em Redes

Neurais Profundas e Inteligência Artificial. Felizmente, houve indivíduos que realizaram a pesquisa sem financiamento.

As primeiras "redes neurais Convolucionais" foram usadas por Kunihiko Fukushima. Fukushima concebeu redes neurais com múltiplas camadas de agrupamento e convoluções. Em 1979, ele desenvolveu uma rede neural artificial, chamada Neocognitron, que usava um design hierárquico e multicamadas. Este design permitiu ao computador "aprender" a reconhecer padrões visuais. As redes se assemelhavam a versões modernas, mas foram treinadas com uma estratégia de reforço de ativação recorrente em múltiplas camadas, que ganhou força ao longo do tempo. Além disso, o design de Fukushima permitiu que os recursos importantes fossem ajustados manualmente aumentando o "peso" de certas conexões.

Muitos dos conceitos de Neocognitron continuam a ser utilizados. O uso de conexões de cima para baixo e novos métodos de aprendizagem permitiram a realização de uma variedade de redes neurais. Quando mais de um padrão é apresentado ao mesmo tempo, o modelo de Atenção Seletiva pode separar e reconhecer padrões individuais deslocando sua atenção de uma para o outro (o mesmo processo que usamos em multitarefa). Um Neocognitron moderno não só pode identificar padrões com informações faltantes (por exemplo, um número 5 desenhado de maneira incompleta), mas também pode completar a imagem adicionando as informações que faltam. Isso pode ser descrito como "inferência".

O Backpropagation, o uso de erros no treinamento de modelos de Deep Learning, evoluiu significativamente em 1970. Foi quando Seppo Linnainmaa escreveu sua tese de mestrado, incluindo um código FORTRAN para Backpropagation. Infelizmente, o conceito não foi aplicado às redes neurais até 1985. Foi quando Rumelhart, Williams e Hinton demonstraram o Backpropagation em uma rede neural que poderia fornecer representações de distribuição "interessantes". Filosoficamente, essa descoberta trouxe à luz a questão dentro da psicologia cognitiva de saber se a compreensão humana depende da lógica simbólica (computacionalismo) ou de representações distribuídas (conexão). Em 1989, YannLecun forneceu a primeira demonstração prática de Backpropagation para ler os dígitos "manuscritos" (assunto do próximo capítulo). Este sistema foi usado para ler o número de cheques manuscritos.

Fig6 – Os pioneiros da Inteligência Artificial. Da esquerda para a direita: Yann LeCun, Geoffrey Hinton, Yoshua Bengio e Andrew Ng.

Porém, tivemos neste período o que ficou conhecido como segundo Inverno da IA, que ocorreu entre 1985-1990, que também afetou pesquisas em Redes Neurais e Aprendizagem Profunda. Vários indivíduos excessivamente otimistas haviam exagerado o potencial "imediato" da Inteligência Artificial, quebrando expectativas e irrtando os investidores. A raiva era tão intensa, que a frase Inteligência Artificial atingiu o status de pseudociência. Felizmente, algumas pessoas continuaram trabalhando com IA e Deep Learning, e alguns avanços significativos foram feitos. Em 1995, Dana Cortes e Vladimir Vapnik desenvolveram a máquina de vetor de suporte ou Upport Vector Machine (um sistema para mapear e reconhecer dados semelhantes). O LSTM (Long-Short Term Memory) para redes neurais recorrentes foi desenvolvido em 1997, por Sepp Hochreiter e Juergen Schmidhuber.

O proximo passo evolutivo significativo para Deep Learning ocorreu em 1999, quando os computadores começaram a se tornar mais rápidos no processamento de dados GPUs (unidades de processamento de gráfico) foram desenvolvidas. O uso de GPUs significou um salto no tempo de processamento, resultando emum almento das velocidades computacionais em 1000 vezes ao longo de um período de 10 anos. Durante esse período, as redes neurais começaram a competir com máquinas de vetor de suporte, as redes neurais ofereciam melhores resultados usanado os mesmos dados. As redes neurais também têm a vantagem de continuar a melhorar à medida que mais dados de treinamento são adicionados.

Em torno do ano 2000, apareceu o problema conhecido como Vanishing Gradient. Foi descoberto que as "características" aprendidas em camadas mais baixas não eram aprendicas pelas camadas superiores, pois nenhum sinal de aprendizado alcançou essas camadas. Este não era um problema fundamental para todas as redes neurais, apenas aquelas com métodos de aprendizagem baseados em gradientes. A origem do problema acabou por ser certas funções de ativação. Uma série de funções de ativação condensavam sua entrada, reduzindo, por sua vez, a faixa de saída de forma extremamente pequena. Nessas áreas de entrada, uma grande mudança será reduzida a uma pequena mudança na saída, resultando em um gradiente em queda. Duas soluçoes utilizadas para resolver este problema foram o pré-treino camada-a-camada e o desenvolvimento de uma memória longa e de curto prazo.

Em 200, um relatório de pesquisa do Grupo META (agora chamado Gartner) descreveu os desafios e oportunidades no crescimento do volume de dados. O relatório descreveu o aumento do volume de dados e a crescente velocidade de dados como o aumento da gama de fontes e tipos de dados. Este foi um apelo para se preparar para a investida do Big Data, que estava apenas começando.

Em 2009, Fei-Fei Li, professora de IA em Stanford na Califórnia, lançou o ImageNet e montou uma base de dados gratuita de mais de 14 milhões de imagens etiquetadas. Eram necessárias imagens marcadas para "treinar" as redes neurais. A professora Li disse: "Nossa visão é que o Big Data mudará a maneira como a aprendizagem de máqina funciona. Data drives learning". Ela acertou em cheio!

Até 2011, a velocidade das GPUs aumentou significativamente, possibilitando a formação de redes convolutivas "sem" o pré-treino camada por camada. Com o aumento da velocidade de computação, tornoi-se óbvio que Deep Learning tinha vantagens significativas em termos de eficiência e volocidade. Um exemplo é a AlexNet, uma rede neutal convolutiva, cuja arquitetura ganhou várias competições internacionais durante 2011 e 2012. As unidades lineares retificadas foram usadas para melhorar a velocidade.

Também em 2012, o Google Brain lançou os resultados de um projeto incomum conhecido como The Cat Experiment. O projeto de espírito livr explorou as dificuldades de "aprendizagem

sem supervisão". A Aprendizagem profunda usa "aprendizagem supervisionada", o que significa que a rede neural convolutiva é treinada usando dados rotulados. Usando a aprendizagem sem supervisão, uma rede neural convolucional é alimentada com dados não marcados, e é então solicitada a busca de padrões recorrentes.

O Cat Experiment usou uma rede neural distribuída por mais de 1.000 computadores. Dez milhões de imagens "sem etiqueta" foram tiradas aleatoriamente do YouTube, mostradas ao sistema e, em seguida, o software de treinamento foi autorizado a ser executado. No final d o treinamento, um neurônio na camada mais alta foi encontrado para responder fortemente às imagens de gatos. Andrew Ng, o fundador do projeto, disse: "Nós tambpem encontramos um neurônio que respondeu fortemente aos rostos humanos". A aprendizagem não supervisionada continua a ser um campo ativo de pesquisa em Aprendizagem Profunda.

Atualmente, o processamento de Big Data e a evolução da Ienteligência Artificial são ambos dependentes da Aprendizagem Profunda. Com Deep Learning podeos construir sistemas inteligentes e estamos nos aproximando da criação de uma IA totalmente autônoma. Isso vai gerar impacto em todos os segmentos da sociedade e aqueles que souberem trabalhar com a tecnologia, serão os líderes desse novo mundo que se apresenta diante de nós.

No próximo capítulo, você vai começar a compreender tecnicamente como funciona a Aprendizagem Profunda. Até o capítulo 4.

Referências:

Deep Learning in a Nutshell: History and Training from NVIDIA

Linnainmaa, S. (1970). The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors. Master's thesis, Univ. Helsinki.

P. Werbos. Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences. PhD thesis, Harvard University, Cambridge, MA, 1974.

Werbos, P.J. (2006). Backwards differentiation in AD and neural nets: Past links and new opportunities. In Automatic Differentiation: Applications, Theory, and Implementations, pages 15-34. Springer.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. Nature, 323, 533–536.

Widrow, B., & Lehr, M. (1990). 30 years of adaptive neural networks: perceptron, madaline, and backpropagation. Proceedings of the IEEE, 78(9), 1415-1442.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. 1986. Learning internal representations by error propagation. In Parallel distributed processing: explorations in the microstructure of cognition, vol. 1, David E. Rumelhart, James L. McClelland, and CORPORATE PDP Research Group (Eds.). MIT Press, Cambridge, MA, USA 318-362

CAPÍTULO 4

O Neurônio, Biológico e Matemático

Para compreender a lógica de funcionamento das redes neurais, alguns conceitos básicos referentes ao funcionamento do cérebro humano e seus componentes, os neurônios, são de fundamental importância. A formação das conexões entre as células e algumas considerações sobre como se concebe teoricamente o funcionamento matemático, ajudam a entender as bases da aprendizagem de máquina e das redes neurais. Vejamos como funciona o neurônio biológico deixando Machine Learning de lado por um instante!

O Neurônio Biológico

O neurônio é a unidade básica do cérebro humano, sendo uma célula especializada na transmissão de informações, pois nelas estão introduzidas propriedades de excitabilidade e condução de mensagens nervosas. O neurônio é constituído por 3 partes principais: a soma ou corpo celular, do qual emanam algumas ramificações denominadas de dendritos, e por uma outra ramificação descendente da soma, porém mais extensa, chamada de axônio. Nas extremidades dos axônios estão os nervos terminais, pelos quais é realizada a transmissão das informações para outros neurônios. Esta transmissão é conhecida como sinapse.

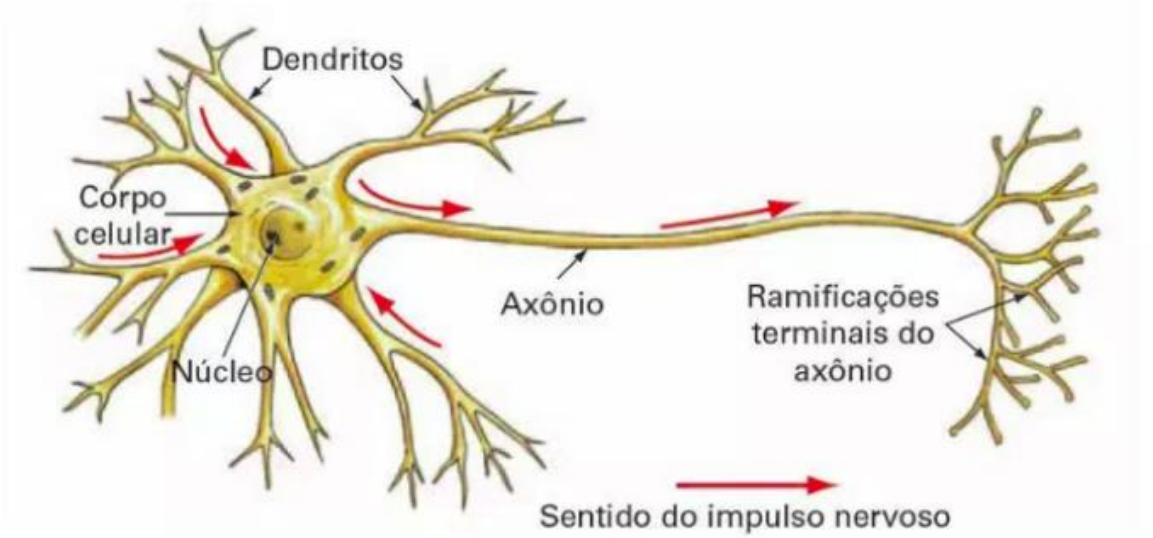


Fig7 – Representação simplificada do Neurônio Biológico

Nosso cérebro é formado por bilhões de neurônios. Mas eles não estão isolados. Pelo contrário, existem centenas de bilhões de conexões entre eles, formando uma enorme rede de comunicação, a rede neural. Cada neurônio possui um corpo central, diversos dendritos e um axônio. Os dendritos recebem sinais elétricos de outros neurônios através das sinapses, que constitui o processo de comunicação entre neurônios. O corpo celular processa a informação e envia para outro neurônio.

Observe que a soma e os dendritos formam a superfície de entrada do neurônio e o axônio a superfície de saída do fluxo de informações (esse fluxo de informação é importante para compreender o neurônio matemático daqui a pouco). A informação transmitida pelos neurônios na realidade são impulsos elétricos. O impulso elétrico é a mensagem que os neurônios transmitem uns aos outros, ou seja, é a propagação de um estímulo ao longo dos neurônios que pode ser qualquer sinal captado pelos receptores nervosos.

Os dendritos têm como função, receber informações, ou impulsos nervosos, oriundos de outros neurônios e conduzi-los até o corpo celular. Ali, a informação é processada e novos impulsos são gerados. Estes impulsos são transmitidos a outros neurônios, passando pelo axônio e atingindo os dendritos dos neurônios seguintes. O corpo do neurônio é responsável por coletar e combinar informações vindas de outros neurônios.

O ponto de contato entre a terminação axônica de um neurônio e o dendrito de outro é chamado de sinapse. É pelas sinapses que os neurônios se unem funcionalmente, formando as redes neurais, As sinapses funcionam como válvulas, sendo capazes de controlar a transmissão de impulsos, isto é, o fluxo de informação entre os neurônios na rede neural. O efeito das sinapses é variável e é esta variação que dá ao neurônio capacidade de adaptação.

Sinais elétricos gerados nos sensores (retina ocular, papilas gustativas, etc...) caminham pelos axônios. Se esses sinais forem superiores a um limiar de disparo (threshold), seguem pelo axônio. Se esses sinais forem superiores a um limiar de disparo (threshold), seguem pelo axônio. Caso contrário, são bloqueados e não prosseguem (são considerados irrelevantes). A passagem desses sinais não é elétrica, mas química (através da substância serotonina). Se o sinal for superior a certo limite (threshold), vai em frente; caso contrário é bloqueado e não segue. Estamos falando aqui do neurônio biológico e preste bastante atenção a palavra threshold, pois ela é a essência do neurônio matemático.

Um neurônio recebe sinais através de inúmeros dendritos, os quais são ponderados e enviados para o axônio, podendo ou não seguir adiante (thresshold). Na passagem por um neurônio, um sinal pode ser amplificado ou atenuado, dependendo do dendrito de origem, pois a cada condutor, está associado um peso pelo qual o sinal é multiplicado. Os pesos são o que chamamos de memória.

Cada região do cérebro é especializada em uma dada função, como processamento de sinais auditivos, sonoros, elaboração de pensamentos, desejos, etc... Esse processamento se dá através de redes particulares interligadas entre si, realizando processamento paralelo. Cada região do cérebro possui uma arquitetura de rede diferente: varia o número de neurônios, de sinapses por neurônio, valor dos thresholds e dos pesos, etc... Os valores dos pesos são estabelecidos por meio de treinamento recebido pelo cérebro durante a vida útil. É a memorização.

Inspirados no neurônio biológico, os pesquisadores desenvolveram um modelo de neurônio matemático que se tornou a base da Inteligência Artificial. A ideia era simples: “Se redes neurais formam a inteligência humana, vamos reproduzir isso e criar inteligência artificial”. E assim nasceu o neurônio matemático, o qual descrevemos abaixo.

O Neurônio Matemático

A partir da estrutura e funcionamento do neurônio biológico, pesquisadores tentaram simular este sistema em computador. O modelo mais bem aceito foi proposto por Warren McCulloch e

Walter Pitts em 1943, o qual implementa de maneira simplificada os componentes e o funcionamento de um neurônio biológico. Em termos simples, um neurônio matemático de uma rede neural artificial é o componente que calcula a soma ponderada de vários inputs, aplica uma função e passa o resultado adiante.

Neste modelo de neurônio matemático, os impulsos elétricos provenientes de outros neurônios são representados pelos chamados sinais de entrada (a letra x nesse diagrama abaixo, que nada mais são do que os dados que alimentam seu modelo de rede neural artificial). Dentre os vários estímulos recebidos, alguns excitarão mais e outros menos o neurônio receptor e essa medida de quão excitatório é o estímulo é representada no modelo de Warren McCulloch e Walter Pitts através dos pesos sinápticos. Quanto maior o valor do peso, mais excitatório é o estímulo. Os pesos sinápticos são representados por wkn neste diagrama abaixo, onde k representa o índice do neurônio em questão e n se refere ao terminal de entrada da sinapse a qual o peso sináptico se refere.

A soma ou corpo da célula é representada por uma composição de dois módulos, o primeiro é uma junção aditiva, somatório dos estímulos (sinais de entrada) multiplicado pelo seu fato excitatório (pesos sinápticos), e posteriormente uma função de ativação, que definirá com base nas entradas e pesos sinápticos, qual será a saída do neurônio. O axônio é aqui representado pela saída (yk) obtida pela aplicação da função de ativação. Assim como no modelo biológico, o estímulo pode ser excitatório ou inibitório, representado pelo peso sináptico positivo ou negativo respectivamente.

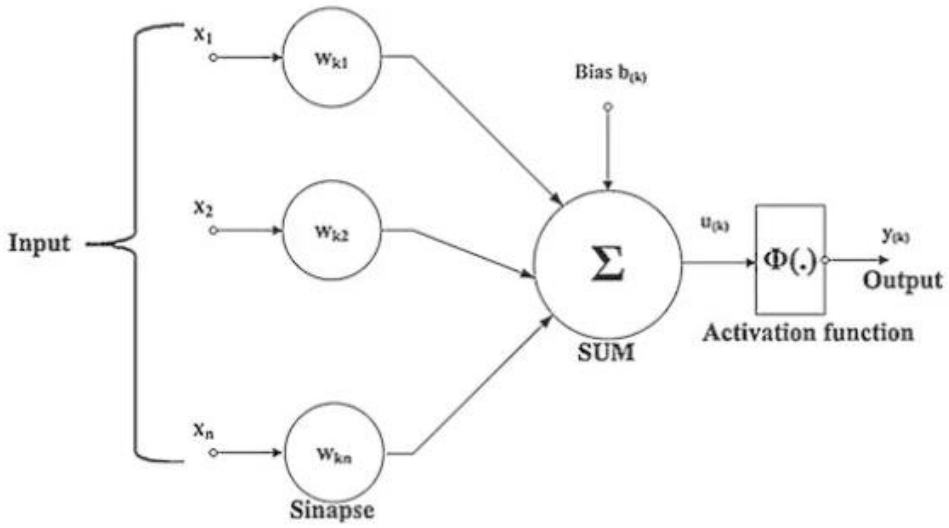


Fig8 – Representação simplificada no Neurônio Matemático

O modelo proposto possui uma natureza binária. Tanto os sinais de entrada quanto a saída, são valores binários. McCulloch acreditava que o funcionamento do sistema nervoso central possuía um caráter binário, ou seja, um neurônio influencia ou não outro neurônio, mas posteriormente mostrou-se que não era dessa forma.

O neurônio matemático é um modelo simplificado do neurônio biológico. Tais modelos inspirados a partir da análise da geração e propagação de impulsos elétricos pela membrana celular dos neurônios. O neurônio matemático recebe um ou mais sinais de entrada e devolve um único sinal de saída, que pode ser distribuído como sinal de saída da rede, ou como sinal de

entrada para um ou vários outros neurônios da camada posterior (que formam a rede neural artificial). Os dendritos e axônios são representados matematicamente apenas pelas sinapses, e a intensidade da ligação é representada por uma grandeza denominada peso sináptico, simbolizada pela letra w, Quando as entradas, x são apresentadas ao neurônio, elas são multiplicadas pelos pesos sinápticos correspondentes, gerando as entradas ponderadas, ou seja, x1 que multiplica w1, etc... Isso descreve uma das bases matemáticas do funcionamento de uma rede neural artificial, a multiplicação de matrizes.

$$\boldsymbol{X}\boldsymbol{w} = \boldsymbol{y}$$

$$\begin{bmatrix} 1 & x_{11} & \dots & x_{1d} \\ 1 & x_{21} & \dots & x_{2d} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_{n1} & \dots & x_{nd} \end{bmatrix} \times \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_d \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}$$

Fig9 – Multiplicação de Matrizes entre Sinais de Entrada x e Pesos Sinápticos w (versão simplificada).

O neurônio então totaliza os produtos gerando um único resultado. A esta função se denomina função de combinação. Este valor é então apresentado a uma função de ativação ou função de transferência, que tem, dentre outras, a finalidade de evitar o acréscimo progressivo de valores de saída ao longo das camadas da rede, visto que tais funções possuem valores máximos e mínimos contidos em intervalos determinados. O uso de funções de transferência não-lineares torna a rede neural uma ferramenta poderosa. Sabe-se que uma rede Perceptron de duas camadas com função de transferência não-linear como a função sigmoide (que veremos mais adiante), é denominada de aproximador universal.

Um neurônio dispara quando a soma dos impulsos que ele recebe ultrapassa o seu mimiar de excitação chamado threshold. O corpo do neurônio, por sua vez, é emulado por um mecanismo simples que faz a soma dos valores xi e wi recebidos pelo neurônio (soma podenderada) e decide se o neurônio deve ou não disparar (saída igual a 1 ou a 0) comparando a soma obtida ao limiar ou threshold do neurônio. A ativação do neurônio é obtida através da aplicação de uma “função de ativação”, que ativa a saída ou não, dependendo do valor da soma ponderada das suas entradas.

Note que este modelo matemático simplificado de um neurônio é estático, ou seja, não considera a dinâmica do neurônio natural. No neurônio biológico, os sinais são enviados em pulsos e alguns componentes dos neurônios biológicos, a exemplo do axônio, funcionam como filtros de frequência.

O modelo do neurônio matemático também pode incluir uma polarização ou bias de entrada. Esta variável e incluída ao somatório da função de ativação, com o intuito de aumentar o grau de liberdade desta função e, consequentemente, a capacidade de aproximação da rede. O valor de bias é ajustado da mesma forma que os pesos sinápticos. O bias possibilita que um neurônio apresente saída não nula ainda que todas as suas entradas sejam nulas. Por exemplo, caso não houvesse o bias e todas as entradas de um neurônio fossem nulas, então o valor da função de ativação seria nulo. Desta forma não poderíamos, por exemplo, fazer com que o neurônio

aprendesse a relação pertinente ao "ou exclusivo" da lógica. Em resumo, temos esses componentes em um neurônio matemático:

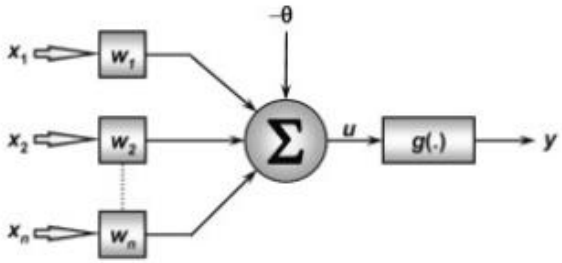


Fig10 – Representação do Neurônio Matemático

- Sinais de entrada { X1, X2, ..., Xn }: São os sinais externos normalmente normalizados para incrementar a eficiência computacional dos algoritmos de aprendizagem. São os dados que alimentam seu modelo preditivo.

- Pesos sinápticos { W1, W2, ..., Wn }: São valores para ponderar os sinais de cada entrada da rede. Esses valores são aprendidos durante o treinamento.

- Combinador linear { $\sum$ }: Agregar todos sinais de entrada que foram ponderados pelos respectivos pesos sinápticos a fim de produzir um potencial de ativação.

- Limiar de ativação { Θ }: Especifica qual será o patamar apropriado para que o resultado produzido pelo combinador linear possa gerar um valor de disparo de ativação.

- Potencial de ativação { ц }: É o resultado obtido pela diferença do valor produzido entre o combinador linear e o de ativação. Se o valor for positivo, ou seja, se $ц > 0$ então o neurônio produz um potencial excitatório; caso contrário, o potencial será inibitório.

- Função de ativação { g }: Seu objetivo é limitar a saída de um neurônio em um intervalo de valores.

- Sinal de saída { y }: É o valor final de saída podendo ser usado coo entrada de outros neurônios que estão sequencialmente interligados.

Os modelos baseados em redes neurais artificiais são os que mais ganharam atenção nos últimos anos por conseguirem resolver problemas de IA nos quais se conseguia pouco avanço com outras técnicas. A partir da concepção do neurônio matemático, várias arquiteturas e modelos com diferentes combinações entre esses neurônios, e aplicando diferentes técnicas matemáticas e estatísticas, surgiram e propiciaram a criação de arquiteturas avançadas de Deep Learning como Redes Neurais Convolucionais, Redes Neurais Recorrentes, Auto Encoders, Generative Adversarial Networks, Memory Networks, entre outras, que estudaremos ao longo deste livro.

Referências:

Anatomia de um Neurônio

Bibliografia Machine Learning e IA

Deep Learning in Neural Networks: An Overview

Grokking Deep Learning

HAYKIN, S. Redes Neurais, princípios e práticas. Porto Alegre: Bookman, 2001.

JAIN, A. K, MAO, J., MOHIUDDIN, K.M. Artificial neural networks: a tutorial. IEEE Computer, v. 29, n. 3, p. 56-63, 1996.

CAPÍTULO 5

Usando Redes Neurais para reconhecer Dígitos Manuscritos

O sistema visual humano é uma das maravilhas do mundo. Considere a seguinte sequência de dígitos manuscritos:

A maioria das pessoas reconhecem sem esforço esses dígitos como 504192. Essa facilidade é enganosa. Em casa hemisfério do nosso cérebro, os seres humanos têm um córtex visual primário, também conhecido como V1, contendo 140 milhões de neurônios, com dezenas de bilhões de conexões entre eles. E, no entanto, a visão humana envolve não apenas V1, mas uma série inteira de córtices visuais – V2, V3, V4 e V5 – fazendo processamento de imagem progressivamente mais complexo. Nós carregamos em nossas cabeças um supercomputador, sintonizado pela evolução ao longo de centenas de milhões de anos, e soberbamente adaptado para entender o mundo visual. Reconhecer os dígitos manuscritos não é fácil. Em vez disso, nós humanos somos estupendos, surpreendentemente bons, em entender o que nossos olhos nos mostram. Mas quase todo esse trabalho é feito inconscientemente. E, portanto, geralmente não apreciamos o quão difícil é o problema dos nossos sistemas visuais.

A dificuldade de reconhecimento do padrão visual torna-se evidente se você tentar escrever um programa de computador para reconhecer dígitos como os acima. O que parece fácil quando nos seres humanos fazemos, de repente, se torna extremamente difícil. Intuições simples sobre como reconhecemos formas – "um 9 tem um loop e um curso vertical no canto inferior direito" não é tão simples de se expressar algoritmicamente. Quando você tenta construir essas regras de forma precisa, você se perde rapidamente em diversas exceções, ressalvas e casos especiais. É meio desesperador.

As redes neurais abordam o problema de uma maneira diferente. A idéia é tomar uma grande quantidade de dígitos manuscritos, conhecidos como exemplos de treinamento, e em seguida, desenvolver um sistema que possa aprender com esses exemplos de treinamento. Em outras palavras, a rede neural usa os exemplos para inferir automaticamente regras para o reconhecimento de dígitos manuscritos. Além disso, ao aumentar o número de exemplos de treinamento, a rede pode aprender mais sobre a caligrafia, e assim melhorar sua precisão. Podemos construir um reconhecedor de dígitos manuscritos melhor usando milhares, milhões ou bilhões de exemplos de treinamento.

Ao longo dos capítulos começaremos nossa jornada rumo às arquiteturas mas avançadas do Deep Learning, desenvolvendo um programa de computador implementando uma rede neural que aprende a reconhecer os dígitos manuscritos. O programa não usará bibliotecas de redes neurais especiais (usaremos apenas linguagem Python). Mas este programa pode reconhecer dígitos com uma precisão de mais de 96%, sem intervenção humana. Além disso, em capítulos posteriores, desenvolveremos ideias que podem melhorar a precisão para mais de 99%. Na verdade, as melhores redes neurais comerciais são agora tão boas que são usadas pelos bancos para processar cheques e por agências de correio para reconhecer endereços.

Estamos nos concentrando no reconhecimento de manuscrito porque é um excelente problema protótipo para aprender sobre redes neurais em geral. Como um protótipo, ele atinge um ponto interessante: é desafiador – não é tão simples reconhecer os dígitos manuscritos – mas também não é tão difícil e nem requer uma solução extremamente complicada, ou um tremendo poder computacional. Além disso, é uma ótima maneira de desenvolver técnicas mais avançadas, como a aprendizagem profunda. E assim, ao longo do livro, retornaremos repetidamente ao problema do reconhecimento de dígitos manuscritos. Mais tarde, no livro, vamos discutir como essas ideias podem ser aplicadas a outros problemas em visão computacional, e também em reconhecimento da fala, processamento de linguagem natural e outras áreas.

Ao longo do caminho, desenvolveremos muitas ideias-chave sobre as redes neurais, incluindo dois tipos importantes de neurônios artificiais (o Perceptron e o neurônio sigmóide) e o algoritmo de aprendizagem padrão para redes neurais, conhecido como descida estocástica do gradiente. Explicaremos porque as coisas são feitas da maneira que elas são e na construção de sua intuição de redes neurais. Isso requer uma discussão mais longa do que apenas apresentar a mecânica básica do que está acontecendo, mas vale a pena para o entendimento mais profundo que você alcançará. E ao final deste livro, você terá uma boa compreensão do que é aprendizado profundo e como isso está transformando o mundo!

Caso você tenha conhecimento em linguagem Python, recomendamos o curso Python Fundamentos para Análise de Dados. Ele vai fornecer uma ótima base de tudo que você precisa para começar a desenvolver suas redes neurais.

Referências:

Machine Learning

The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition

Pattern Recognition and Machine Learning

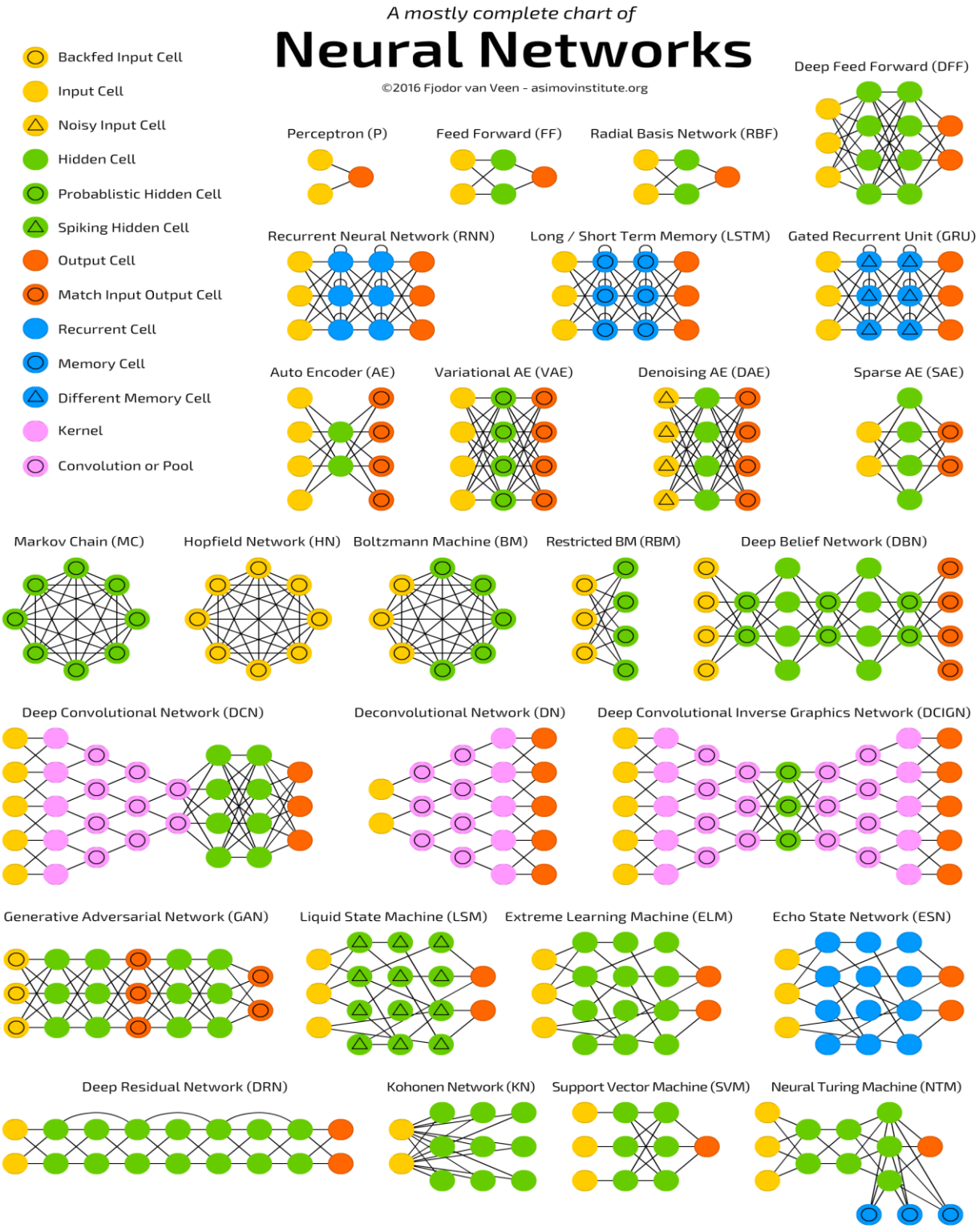
Redes Neurais, princípios e práticas

Neural Networks and Deep Learning (alguns trechos extraídos e traduzidos com autorização do autor Michael Nielsen)

CAPÍTULO 6

O Perceptron – Parte 1

Você sabe quais são as principais arquiteturas de rede neurais artificiais? Não. Então analise cuidadosamente a imagem abaixo (excelente trabalho criado pela equipe do Asimov Institute, cujo link você encontra na seção de referências ao final deste capítulo).

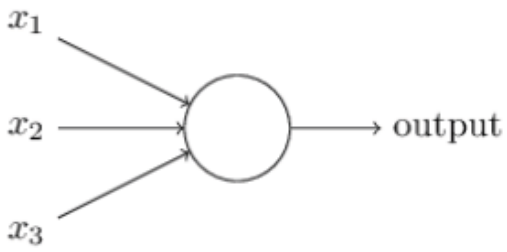


Incrível, não? São diversas arquiteturas, usadas para resolver diferentes tipos de problemas, como por exemplo as arquiteturas de redes neurais convolucionais usadas em problemas de Visão Computacional e as redes neurais recorrentes usadas em problemas de Linguagem Natural. Estudaremos quase todas essas arquiteturas aqui neste livro. Sim, isso mesmo que você leu. Estamos apenas começando!!

Embora todas essas arquiteturas sejam de redes neurais artificiais, nem todas são de Deep Learning. O que caracteriza modelos de aprendizagem profunda, como o nome sugere, são redes neurais artificiais com muitas camadas ocultas (ou intermediárias). Mas antes de chegarmos lá, precisamos passar pela arquitetura mais simples de uma rede neural artificial, o Perceptron. Como diz o ditado: "Toda grande caminhada começa pelo primeiro passo".

O Modelo Perceptron foi desenvolvido nas décadas de 1950 e 1960 pelo cientista Frank Rosenblatt, inspirado em trabalhos anteriores de Warren McCulloch e Walter Pitts. Hoje, é mais comum usar outros modelos de neurônios artificiais, mas o Perceptron perminte uma compreensão clara de como funciona uma rede neural em termos metemáticos, sendo uma excelente introdução.

Então, como funcionam os Perceptrons? Um Perceptron é um modelo matemático que recebe várias entradas, x1, x2, ... e produz uma única saída binária:



No exemplo mostrado, o Perceptron possui três entradas: x1, x2, x3. Rosenblatt propôs uma regra simples para calcular a saída. Ele introduziu pesos, w1, w2, ..., números reais expressando a importância das respectivas entradas para a saída. A saída do neurônio, 0 ou 1, é determinada pela soma ponderada, **∑jwjxj,** menor ou maior do que algum valor limiar (threshold). Assim como os pesos, o threshold é um número real que é um parâmetro do neurônio. Para colocá-lo em termos algébricos mais precisos:

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases}$$

Esse é o modelo matemático básico. Uma meneira de pensar sobre o Perceptron é que é um dispositivo que toma decisões ao comprovar evidências. Deixe-me dar um exemplo. Não é um exemplo muito realista, mas é fácil de entender, e logo chegaremos a exemplos mais realistas. Suponha que o fim de semana esteja chegando e você ouviu falar que haverá um festival de queijo em sua cidade. Você gosta de queijo e está tentando decidir se deve ou não ir ao festival. Você pode tomar sua decisão pensando três fatores:

- O tempo está bom?
- Seu namorado ou namorada quer acompanhá-lo(a)?
- O festival está perto de transporte público? (Você não possui um carro)

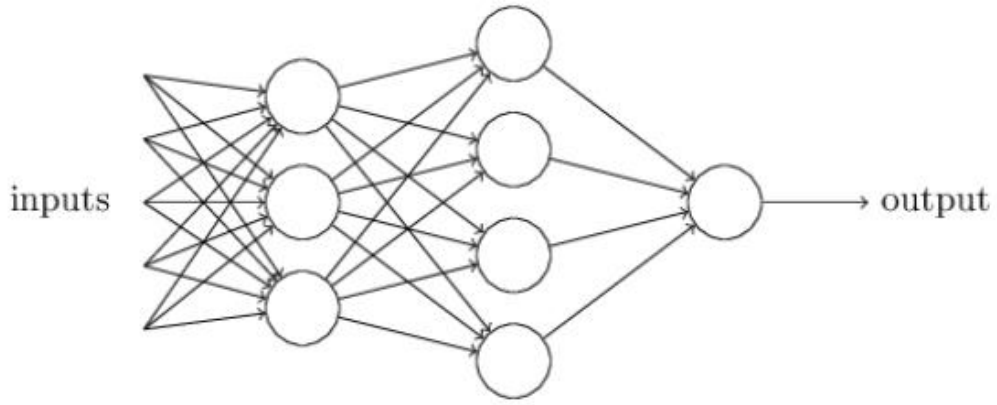
Podemos representar estes três fatores pelas variáveis binárias correspondentes x1, x2 e x3. Por exemplo, teríamos $x1 = 1$ se o tempo estiver bom e $x1 = 0$ se o tempo estiver ruim. Da mesma forma, $x2 = 1$ se seu namorado ou namorada quiser ir ao festival com você, e $x2 = 0$, se não. E similarmente para x3 e transporte público.

Agora, suponha que você adore queijo e está disposto a ir ao festival, mesmo que seu namorado ou namorada não esteja interessado e o festival fica em um lugar de difícil acesso e sem transporte público amplamente disponível. Além disso, você realmente detesta mau tempo, e não há como ir ao festival se o tempo estiver ruim. Você pode usar Perceptrons para modelar esse tipo de tomada de decisão.

Uma maneira de fazer isso é escolher um peso $w1 = 6$ para o tempo e $w2 = 2$ para as outras condições. O valor maior de w1 indica que o tempo é muito importante para você, muito mais do que se seu namorado ou namorada vai acompanhá-lo(a) ou se o festival é próximo do transporte público. Finalmente, suponha que você escolha um threshold de 5 para o Perceptron. Com essas escolhas, o Perceptron implementa o modelo de tomada de decisão desejado, produzindo 1 sempre que o tempo estiver bom e 0 sempre que o tempo estiver ruim. Não faz diferença para o resultado se seu namorado ou namorada quer ir, ou se o transporte público está acessível.

Variando os pesos e o limiar, podemos obter diferentes modelos de tomada de decisão, Por exemplo, suponha que escolhemos um threshold de 3. Então, o Perceptron decidirá que você deveria ir ao festival sempre que o tempo estiver bom ou quando o festival estiver perto do transporte público e seu namorado ou namorada estiver disposto a se juntar a você. Em outras palavras, seria um modelo diferente de tomada de decisão. Reduzir o threshold significa que você está mais propenso a ir ao festival.

Obviamente, o Perceptron não é um modelo completo de tomada de decisão humana! Mas o que o exemplo ilustra é como um Perceptron pode pesar diferentes tipos de evidências para tomar decisões. E deve parecer plausível que uma rede complexa de Perceptrons possa tomar decisões bastante sutis.



Nesta rede, a primeira coluna de Perceptrons – o que chamaremos de primeira camada de Perceptrons – está tomando três decisões muito simples, pesando a evidência de entrada. E quanto aos Perceptrons na segunda camada? Cada um desses Perceptrons está tomando uma decisão ponderando os resultados da primeira camada de tomada de decisão. Desta forma, um Perceptron na segunda camada pode tomar uma decisão em um nível mais complexo e mais abstrato do que os Perceptrons na primeira camada. E as decisões ainda mais complexas podem ser feitas pelos Perceptrons na terceira camada. Desta forma, uma rede de Perceptrons de várias camadas pode envolver-se em uma tomada de decisão sofisticada.

Aliás, quando definimos os Perceptrons, dissemos que um Perceptron possui apenas uma saída. Na rede acima, os Perceptrons parecem ter múltiplos resultados. Na verdade, eles ainda são de saída única. As setas de saída múltiplas são meramente uma maneira útil de indicar que a saída de um Perceptron está sendo usada como entrada para vários outros Perceptrons.

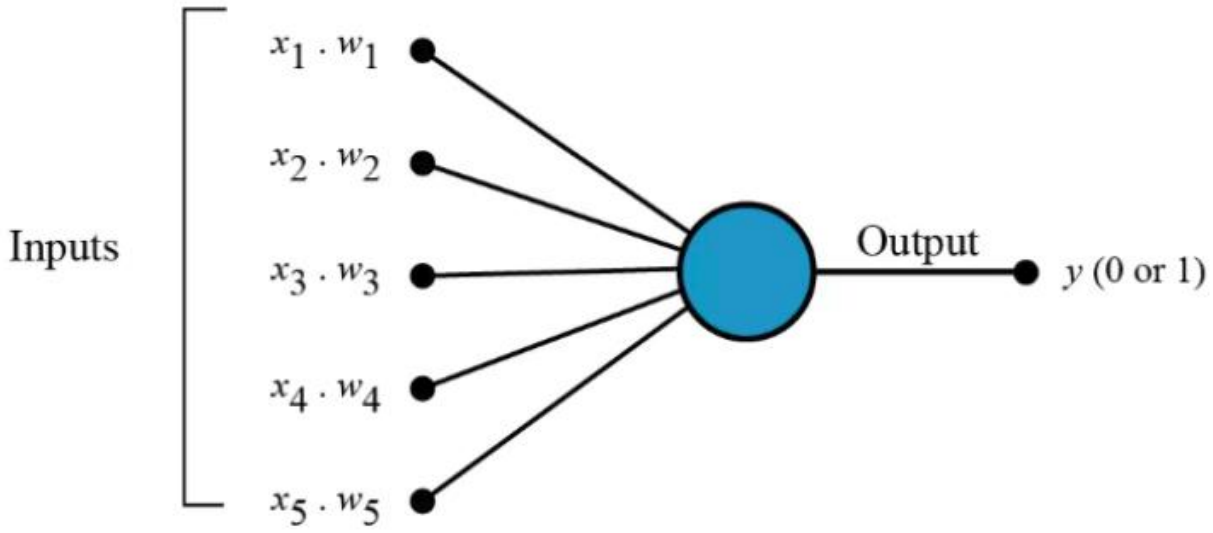
Vamos simplificar a maneira como descrevemos os Perceptrons. No limite de condição $\sum_j w_j x_j > \text{threshold}$ podemos fazer duas mudanças de notação para simplificá-lo. A primeira mudança é escrever $\sum_j w_j x_j$ como um produto (dot product), $w \cdot x = \sum_j w_j x_j$, onde w e x são vetores cujos componentes são os pesos e entradas, respectivamente. A segunda mudança é mover o threshold para o outro lado da equação e substituí-lo pelo que é conhecido como o viés (bias) do Perceptron, ou $b = -\text{threshold}$. Usando o viés em vez do threshold, a regra Perceptron pode ser reescrita:

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

Você pode pensar no viés como uma medida de quão fácil é obter o Perceptron para produzir um 1. Ou para colocá-lo em termos mais biológicos, o viés é uma medida de quão fácil é fazer com que o Perceptron dispare. Para um Perceptron com um viés realmente grande, é extremamente fácil para o Perceptron emitir um 1. Mas se o viés é muito negativo, então é difícil para o Perceptron emitir um 1. Obviamente, a introdução do viés é apenas uma pequena mudança em como descrevemos Perceptrons, mas veremos mais adiante que isso leva a outras

simplificações de notação. Por isso, no restante do livro, não usaremos o threshold, usaremos sempre viés.

Agora começa a ficar mais fácil compreender o conceito por trás das redes neurais artificiais e isso será muito útil quando estudarmos arquiteturas mais avançadas! Um Perceptron segue o modelo "feed-foward", o que significa que as entradas são enviadas para o neurônio, processadas e resultam em uma saída. No diagrama abaixo, isso significa que a rede (um neurônio) lê da esquerda para a direita.



O processo de treinamento de um modelo Perceptron consiste em fazer com que o modelo aprenda os valores ideais de pesos e bias. Apresentamos ao modelo os dados de entrada e as possíveis saídas, treinamos o modelo e pesos e bias são aprendidos. Com o modelo treinado, podemos apresentar novos dados de entrada e o modelo será capaz de prever a saída. Veremos isso em breve quando criarmos nosso primeiro modelo usando linguagem Python.

Perceptron é uma rede neural de camada única e um Perceptron de várias camadas é chamado de Rede Neural Artificial. O Perceptron é um classificador linear (binário). Além disso, é usado na aprendizagem supervisionada e pode ser usado para classificar os dados de entrada fornecidos.

Mas o Perceptron tem ainda outras características importantes, como a representação de condicionais lógicos (and, or, xor), problemas com dados não linearmente separáveis e as funções de ativação. Mas esses são temas para o próximo capítulo. Até lá!

Referências:

The Neural Network Zoo

Machine Learning

The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition

Pattern Recognition and Machine Learning

Redes Neurais, princípios e práticas

Neural Networks and Deep Learning (alguns trechos extraídos e traduzidos com autorização do autor Michael Nielsen)

CAPÍTULO 7

O Perceptron – Parte 2

O Perceptron é um modelo matemático de um neurônio biológico. Enquanto nos neurônios reais o dendrito recebe sinais elétricos dos axônios de outros neurônios, no Perceptron estes sinais elétricos são representados como valores numéricos. Nas sinapses entre dendritos e axônio, os sinais elétricos são modulados em várias quantidades. Isso também é modelado no Perceptron multiplicando cada valor de entrada por um valor chamado peso. Um neurônio real dispara um sinal de saída somente quando a força total dos sinais de entrada excede um certo limiar. Nós modelamos esse fenômeno em um Perceptron calculando a soma ponderada das entradas para representar a força total dos sinais de entrada e aplicando uma função de ativação na soma para determinar sua saída. Tal como nas redes neurais biológicas, esta saída é alimentada em outros Perceptrons. Estudamos tudo isso no capítulo anterior. Agora vamos continuar nossa discussão sobre o Perceptron compreendendo mais alguns conceitos, que serão fundamentais mais a frente quando estudarmos as arquiteturas de Deep Learning.

Antes de iniciar, vamos definir dois conceitos que você vai ver com frequência daqui em diante, vetor de entrada e vetor de pesos:

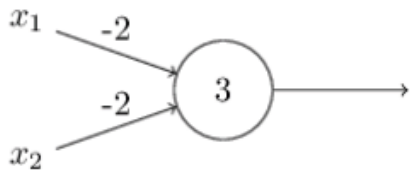
Vetor de entrada – todos os valores de entrada de cada Perceptron são coletivamente chamados de vetor de entrada desse Perceptron. Esses são seus dados de entrada.

Vetor de pesos – de forma semelhante, todos os valores de peso de cada Perceptron são coletivamente chamados de vetor de peso desse Perceptron. Iniciamos nossa rede neural artificial com valores aleatórios de pesos e durante o treinamento a rede neural aprende os valores de peso ideais. Como veremos, existem muitas formas de realizar esse processo.

Boa parte do trabalho de uma rede neural vai girar em torno das operações algébricas entre o vetor de entrada e o vetor de pesos. Em seguida, vamos adicionando outras camadas matemáticas ou estatísticas para realizar diferentes operações, de acordo com o problema que estamos tentando resolver com o modelo de rede neural. Você vai perceber que tudo não passa de Matemática, que pode ser implementada com linguagens de programação, grandes conjuntos de dados de processamento paralelo, para formar sistemas de Inteligência Artificial.

Mas o que um Perceptron pode fazer afinal?

No capítulo anterior descrevemos os Perceptrons como um método para pesar evidencias a fim de tomar decisões. Outra forma em que os Perceptrons podem ser usados é para calcular as funções lógicas elementares tais como AND, OR e NAND (caso tenha dúvidas sobre as operações lógicas, consulte as referências ao final deste capítulo). Por exemplo, suponha que tenhamos um Perceptron com duas entradas, cada uma com peso -2 e um viés de 3. Aqui está nosso Perceptron:



Então vemos que a entrada 00 produziria a saída 1, uma vez que (-2) * 0 + (-2) * 0 + 3 = 3, é positivo (resultado positivo, gera saída 1 do Perceptron, lembra do capítulo anterior?). Aqui, incluímos o símbolo * para tornar as multiplicações explícitas. Cálculos similares mostram que as entradas 01 e 10 produzem a saída 1. Mas a entrada 11 produz a saída 0, uma vez que (-2) * 1 + (-2) * 1 + 3 = -1, é negativo. E assim nosso Perceptron implementa um "portão" NAND, ou uma operação lógica binária NAND.

O exemplo NAND mostra que podemos usar Perceptrons para calcular funções lógicas simples. Na verdade, podemos usar redes de Perceptrons para calcular qualquer função lógica. A razão é que o portão NAND é universal para computação, ou seja, podemos construir qualquer computação com portões NAND.

Uma rede de Perceptrons pode ser usada para simular um circuito contendo muitos portões NAND. E como os portões NAND são universais para a computação, segue-se que os Perceptrons também são universais para a computação. Considerando que o Perceptron é o modelo mais simples de rede neural, imagine o que pode ser feito com modelos bem mais avançados! Acertou se você pensou em Inteligência Artificial.

A universalidade computacional dos Perceptrons é simultaneamente reconfortante e decepcionante. É reconfortante porque nos diz que redes de Perceptrons podem ser tão poderosas como qualquer outro dispositivo de computação. Mas também é decepcionante, porque parece que os Perceptrons são meramente um novo tipo de portão NAND. Isso não é uma grande notícia!

No entanto, a situação é melhor do que esta visão sugere. Acontece que podemos conceber algoritmos de aprendizado que podem ajustar automaticamente os pesos e os vieses de uma rede de neurônios artificiais. Este ajuste ocorre em resposta a estímulos externos, sem intervenção direta de um programador. Esses algoritmos de aprendizagem nos permitem usar neurônios artificiais de uma maneira que é radicalmente diferente dos portões lógicos convencionais. Em vez de colocar explicitamente um circuito de NAND e outros portões, nossas redes neurais podem simplesmente aprender a resolver problemas, às vezes problemas em que seriam extremamente difíceis de projetar diretamente usando um circuito convencional de lógica.

Operações Lógicas e Regiões Linearmente Separáveis

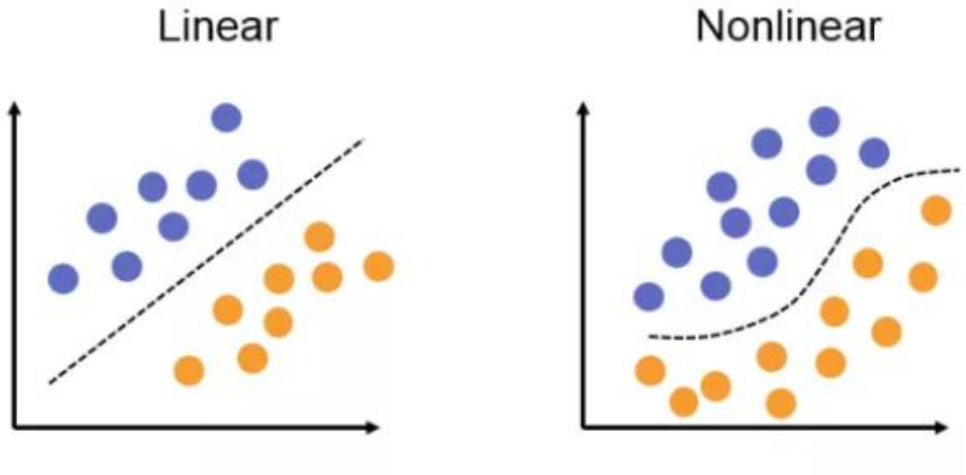
Conforme mencionado acima, um Perceptron calcula a soma ponderada dos valores de entrada. Por simplicidade, suponhamos que existem dois valores de entrada, x e y para um certo Perceptron P. vamos definir os pesos de x e y, como sendo A e B, respectivamente. A soma ponderada pode ser representada como: A x + B y.

Uma vez que o Perceptron produz um valor não-zero somente quando a soma ponderada excede um certo limite C, pode-se escrever a saída deste Perceptron da seguinte maneira:

$$\text{Output de P} = \begin{cases} 1 \text{ se } A\,x + B\,y > C \\ 0 \text{ se } A\,x + B\,y <= C \end{cases}$$

Considerando que $A\,x + B\,y > C$ e $A\,x + B\,y < C$ são as duas regiões no plano xy separadas pela linha $A\,x + B\,y + C = 0$, e se considerarmos ainda a entrada (x, y) como um ponto em um plano, então o Perceptron realmente nos diz qual região no plano a que esse ponto pertence. Tais regiões, uma vez que são separadas por uma única linha, são chamadas de regiões linearmente separáveis.

Um único Perceptron consegue resolver somente funções linearmente separáveis. Em funções não linearmente separáveis, o Perceptron não consegue gerar um hiperplano, esta linha nos gráficos abaixo, para separar os dados. A questão é que no mundo real raramente os dados são linearmente separáveis, fazendo com o que o Perceptron não seja muito útil para atividades práticas (mas sendo ideal para iniciar o estudo em redes neurais artificiais). E como separamos os dados não linearmente separáveis? Continue acompanhando este livro e você irá descobrir.



Mas ainda assim o Perceptron tem sua utilidade, porque resulta em algumas funções lógicas, como os operadores booleanos AND, OR e NOT, que são linearmente separáveis, isto é, eles podem ser realizados usando um único Perceptron. Podemos ilustrar porque eles são linearmente separáveis ao traçar cada um deles em um gráfico:

Nos gráficos acima, os dois eixos são as entradas que podem levar o valor de 0 ou 1 e os números no gráfico são a saída esperada para uma entrada específica. Usando um vetor de peso apropriado para cada caso, um único Perceptron pode executar todas essas funções.

No entanto, nem todos os operadores de lógica são linearmente separáveis. Por exemplo, o operador XOR não é linearmente separável e não pode ser alcançado por um único Perceptron. No entanto, esse problema poderia ser superado usando mais de um Perceptron organizado em redes neurais feed-foward, que veremos mais a frente nos próximos capítulos.

1	1	0
0	0	1
XOR	0	1

Uma vez que é impossível desenhar uma linha para dividir as regiões contendo 1 ou 0, a função XOR não é linearmente separável conforme pode ser visto no gráfico acima.

Agora fica mais fácil compreender porque precisamos de arquiteturas mais avançadas de redes neurais artificiais, uma vez que temos problemas complexos no mundo real, como Visão Computacional, Processamento de Linguagem Natural, Tradução, Detecção de Fraudes, Classificação e muitos outros. E veremos essas arquiteturas em detalhes. Mas antes, precisamos falar sobre um componente fundamental das redes neurais, a Função de Ativação. No próximo capítulo! Até lá!

Referências:

Operação Lógica AND

Operação Lógica OR

Operação Lógica NAND

Operação Lógica XOR

Machine Learning

The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition

Pattern Recognition and Machine Learning

Redes Neurais, princípios e práticas

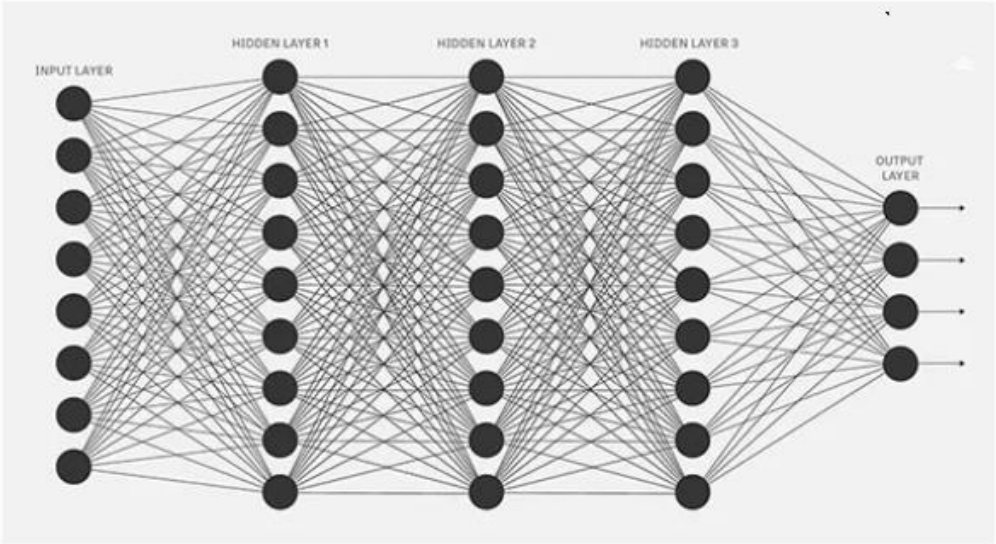
Neural Networks and Deep Learning (alguns trechos extraídos e traduzidos com autorização do autor Michael Nielsen)

CAPÍTULO 8

Funcão de Ativação

Neste capítulo estudaremos um importante componente de uma rede neural artificial, a Funcão de Ativação. Este capítulo é uma introdução ao tema e voltaremos a ele mais adiante quando estudarmos as arquiteturas avançadas de Deep Learning. Esse capítulo pode ser um pouco desafiador, pois começaremos a introduzir conceitos mais avançados, que serão muito úteis na sequência dos capítulos. Relaxe, faça a leitura e aprenda um pouco mais sobre redes neurais artificiais.

Antes de mergulhar nos detalhes das funções de ativação, vamos fazer uma pequena revisão do que são redes neurais artificiais e como funcionam. Uma rede neural é um mecanismo de aprendizado de máquina (Machine Learning) muito poderoso que imita basicamente como um cérebro humano aprende. O cérebro recebe o estímulo do mundo exterior, faz o processamento e gera o resultado. À medida que a tarefa se torna complicada, vários neurônios formam uma rede complexa, transmitindo informações entre si. Usando uma rede neural artificial, tentamos imitar um comportamento semelhante. A rede que você vê abaixo é uma rede neural artificial composta de neurônios interligados.

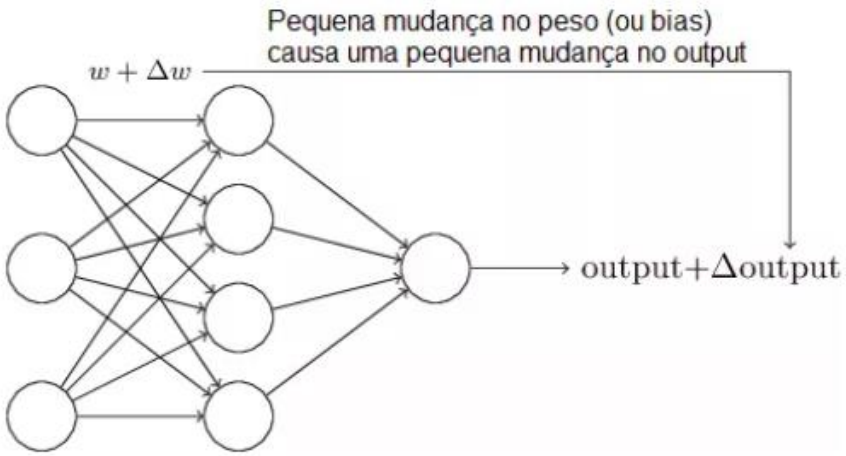


Os círculos negros na imagem acima são neurônios. Cada neurônio é caracterizado pelo peso, bias e a função de ativação. Os dados de entrada são alimentados na camada de entrada. Os neurônios fazem uma transformação linear na entrada pelos pesos e bias. A transformação não linear é feita pela função de ativação. A informação se move da camada de entrada para as camadas ocultas. As camadas ocultas fazem o processamento e enviam a saída final para a camada de saída. Este é o movimento direto da informação conhecido como propagação direta. Mas e se o resultado gerado estiver longe do valor esperado? Em uma rede neural, atualizaríamos os pesos e bias dos neurônios com base no erro. Este processo é conhecido como Backpropagation. Uma vez que todos os dados passaram por este processo, os pesos e bias são usados para previsões.

Calma, calma, calma. Muita informação em um único parágrafo, eu sei! Vamos por partes. As entradas, os pesos e bias nós já discutimos nos capítulos anteriores. A função de ativação vamos discutir agora e a propagação direta e o Backpropagation discutimos nos próximos capítulos!

Função de Ativação

Os algoritmos de aprendizagem são fantásticos. Mas como podemos elaborar esses algoritmos para uma rede neural artificial? Suponhamos que tenhamos uma rede de Perceptrons que gostaríamos de usar para aprender a resolver algum problema. Por exemplo, as entradas para a rede poderiam ser os dados de pixel de uma imagem digitalizada, escrita à mão, de um dígito. Gostaríamos que a rede aprendesse pesos e bias para que a saída da rede classifique corretamente o dígito. Para ver como a aprendizagem pode funcionar, suponha que façamos uma pequena alteração em algum peso (ou bias) na rede. O que queremos é que esta pequena mudança de peso cause apenas uma pequena alteração correspondente na saída da rede. Como veremos em um momento, esta propriedade tornará possível a aprendizagem. Esquematicamente, aqui está o que queremos (obviamente, esta rede é muito simples para fazer reconhecimento de escrita, mas fique tranquilo que veremos redes bem mais complexas).



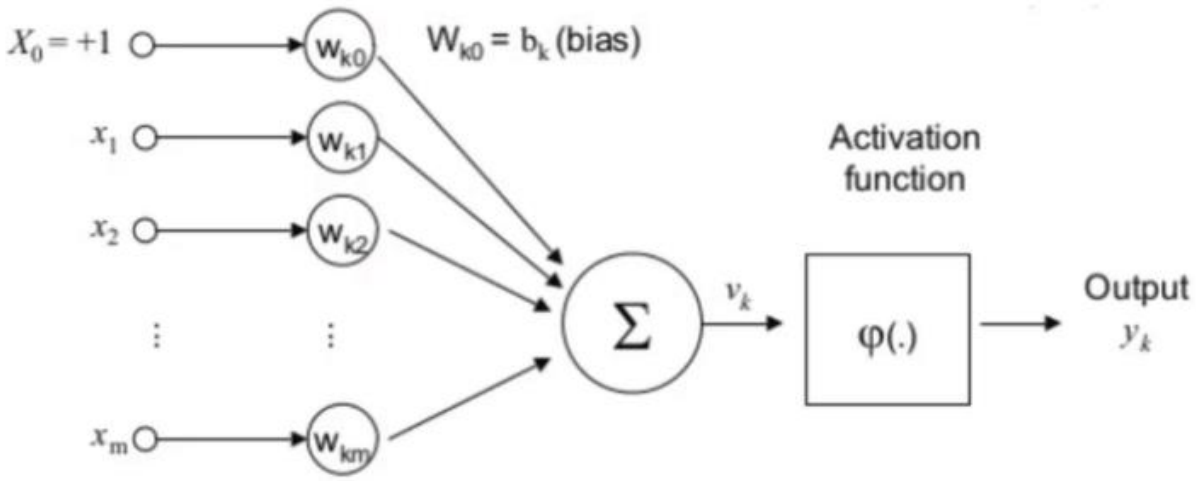
Se fosse verdade que uma pequena alteração em um peso (ou bias) fizesse com que tivéssemos apenas uma pequena alteração no resultado, então poderíamos usar esse fato para modificar os pesos e os valores de bias para que a nossa rede se comporte mais da maneira que queremos. Por exemplo, suponha que a rede classifique equivocadamente uma imagem como “8” quando deveria ser um “9”. Podemos descobrir como fazer uma pequena mudança nos pesos e bias para que a rede fique um pouco mais próxima da classificação da imagem como “9”. E então, repetiríamos isso, mudando os pesos e os valores de bias repetidamente para produzir melhor e melhor resultado. A rede estaria aprendendo.

O problema é que isso não é o que acontece quando nossa rede contém apenas Perceptrons, conforme estudamos nos capítulos anteriores. De fato, uma pequena alteração nos pesos de um único Perceptron na rede pode, por vezes, fazer com que a saída desse Perceptron mude completamente, digamos de 0 a 1. Essa mudança pode então modificar o comportamento do resto da rede, alterando-a completamente de uma maneira muito complicada. Então, enquanto o seu “9” pode agora ser classificado corretamente, o comportamento da rede em todas as

outras imagens provavelmente mudará completamente de maneira difícil de controlar. Talvez haja uma maneira inteligente de resolver esse problema. Sim, há. E é conhecida como função de ativação.

Podemos superar esse problema através da introdução de um componente matemático em nosso neurônio, chamado função de ativação. As funções de ativação permitem que pequenas mudanças nos pesos e bias causem apenas uma pequena alteração no output. Esse é o fato crucial que permitirá que uma rede de neurônios artificiais aprenda.

Vejamos como isso funciona:



As funções de ativação são um elemento extremamente importante das redes neurais artificiais. Elas basicamente decidem se um neurônio deve ser ativado ou não. Ou seja, se a informação que o neurônio está recebendo é relevante para a informação fornecida ou deve ser ignorada. Veja na fórmula abaixo como a função de ativação é mais uma camada matemática no processamento.

$$Y = \text{Activation}(\Sigma(weight * input) + bias)$$

A função de ativação é a transformação não linear que fazemos ao longo do sinal de entrada. Esta saída transformada é então enviada para a próxima camada de neurônios como entrada. Quando não temos a função de ativação, os pesos e bias simplesmente fazem uma transformação linear. Uma equação linear é simples de resolver, mas é limitada na sua capacidade de resolver problemas complexos. Uma rede neural sem função de ativação é essencialmente apenas um modelo de regressão linear. A função de ativação faz a transformação não-linear nos dados de entrada, tornando-o capaz de aprender e executar tarefas mais complexas. Queremos que nossas redes neurais funcionem em tarefas complicadas, como traduções de idiomas (Processamento de Linguagem Natural) e classificações de imagens (Visão Computacional). As transformações lineares nunca seriam capazes de executar tais tarefas.

As funções de ativação tornam possível a propagação posterior desde que os gradientes sejam fornecidos juntamente com o erro para atualizar os pesos e bias. Sem a função não linear diferenciável, isso não seria possível. Caso o termo gradiente não seja familiar, aguarde os próximos capítulos, quando vamos explicar este conceito em detalhes, visto que ele é a essência do processo de aprendizagem em redes neurais artificiais.

Mas não existe apenas um tipo de função de ativação. Na verdade, existem vários, cada qual a ser usado em diferentes situações. Vamos a uma breve descrição dos tipos mais populares.

Tipos Populares de Funções de Ativação

A função de ativação é um componente matemático incluído na estrutura de redes neurais artificiais a fim de permitir a solução de problemas complexos. Existem diversos tipos de funções de ativação e esta é uma área de pesquisa ativa, à medida que a inteligência Artificial evolui (não é maravilhoso estar participando desta evolução, que vai transformar completamente o mundo?). Vejamos quais os tipos mais populares.

Funcão de Etapa Binária (Binary Step Function)

A primeira coisa que vem à nossa mente quando tempos uma função de ativação seria um classificador baseado em limiar (threshold), ou seja, se o neurônio deve ou não ser ativado. Se o valor Y estiver acima de um valor de limite determinado, ative o neurônio senão deixa desativado. Simples! Essa seria a regra:

$$f(x) = 1, x >= 0$$

$$f(x) = 0, x < 0$$

A função de etapa binária é isso mesmo, extremamente simples. Ela pode ser usada ao criar um classificador binário. Quando simplesmente precisamos dizer sim ou não para uma única classe, a função de etapa seria a melhor escolha, pois ativaria o neurônio ou deixaria zero.

A função é mais teórica do que prática, pois, na maioria dos casos, classificamos os dados em várias classes do que apenas uma única classe. A função de etapa não seria capaz de fazer isso.

Além disso, o gradiente da função de etapa é zero. Isso faz com que a função de etapa não seja tão útil durante o Backpropagation quando os gradientes das funções de ativação são enviados para cálculos de erro para melhorar e otimizar os resultados. O gradiente da função de etapa reduz tudo para zero e a melhoria dos modelos realmente não acontece. Lembrando, mais uma vez, que veremos em detalhes os conceitos de gradiente e Backpropagation mais adiante, nos próximos capítulos!

Função Linear

Nós vimos o problema com a função step, o gradiente sendo zero, é impossível atualizar o gradiente durante a Backpropagation. Em vez de uma função de passo simples, podemos tentar usar uma função linear. Podemos definir a função como:

$f(x) = ax$

A derivada de uma função linear é constante, isto é, não depende do valor de entrada x. Isso significa que toda vez que fazemos Backpropagation, o gradiente seria o mesmo. E este é um grande problema, não estamos realmente melhorando o erro, já que o gradiente é praticamente o mesmo. E não apenas suponha que estamos tentando realizar uma tarefa complicada para a qual precisamos de múltiplas camadas em nossa rede. Agora, se cada camada tiver uma transformação linear, não importa quantas camadas nós tenhamos, a saída final não é senão uma transformação linear de entrada. Portanto, a função linear pode ser ideal para tarefas simples, onde a interpretabilidade é altamente desejada.

Sigmóide

Sigmóide é uma função de ativação amplamente utilizada. É da forma:

$f(x) = 1 / (1 + e^{-x})$

Esta é uma função suave e é continuamente diferenciável. A maior vantagem sobre a função de etapa e a função linear é que não é linear. Esta é uma característica incrivelmente interessante da função sigmóide. Isto significa essencialmente que quando eu tenho vários neurônios com função de ativação – a saída também não é linear. A função varia de 0 a 1 tendo um formato S.

A função essencialmente tenta empurrar os valores de y para os extremos. Esta é uma qualidade muito desejável quando tentamos classificar os valores para uma classe específica.

A função sigmoide ainda é amplamente utilizada até hoje, mas ainda temos problemas que precisamos abordar. Com a sigmoide temos problemas quando os gradientes se tornam muito pequenos. Isso significa que o gradiente está se aproximando de zero e a rede n]ao está realmente aprendendo.

Outro problema que a função sigmoide sofre é que os valores variam apenas de 0 a 1. Esta medida que a função sigmóide não é simétrica em torno da origem e os valores recebidos são todos positivos. Nem sempre desejamos que os valores enviados ao próximo neurônio sejam todos do mesmo sinal. Isso pode ser abordado pela ampliação da função sigmoide. Isso é exatamente o que acontece na função tanh.

Tanh

A função tanh é muito semelhante à função sigmoide. Na verdade, é apenas uma versão escalonada da função sigmoide.

Tanh (x) = 2sigmoides (2x) -1

Pode ser escrito diretamente como:

tanh (x) = 2 / (1 + e^(-2x)) -1

Tanh funciona de forma semelhante à função sigmoide, mas sim simétrico em relação â origem, varia de -1 a 1.

Basicamente, soluciona o nosso problema dos valores, sendo todos do mesmo sinal. Todas as outras propriedades são as mesmas da função sigmoide. É contínuo e diferenciável em todos os pontos. A função não linear, então podemos fazer o Backpropagation facilmente nos erros.

ReLU

A função ReLU é a unidade retificada. É definida como:

f(x) = max (0, x)

ReLU é a função de ativação mais amplamente utilizada ao projetar redes neurais atualmente. Primeiramente, a função ReLU é não linear, o que significa que podemos facilmente copiar os erros para trás e ter várias camadas de neurônios ativados pela função ReLU.

A principal vantagem de usar a função ReLU sobre outras funções de ativação é que ela não ativa todos os neurônios ao mesmo tempo. O que isto significa? Se você olhar para a função ReLU e a entrada for negativa, ela será convertida em zero e o neurônio não será ativado. Isso significa que, ao mesmo tempo, apenas alguns neurônios são ativados, tornando a rede esparsa e eficiente e fácil para a computação.

Ma ReLU também pode ter problemas com gradientes que se deslocam em direção a zero. Mas quando temos um problema, sempre podemos pensar em uma solução. Aliás, isso é o que as empresas mais procuram nos dias de hoje: “resolvedores de problemas”. Seja um e sua empregabilidade estará garantida!

Leaky ReLU

A função Leaky ReLU não passa de uma versão melhorada da função ReLU. Na função ReLU, o gradiente é o 0 para $x < 0$, o que faz os neurônios morrerem por ativações nessa região. Leaky

ReLU ajuda a resolver este problema. Em vez de definir a função ReLU como 0 para x inferior a 0, definimos como um pequeno componente linear de x. Pode ser definido como:

$f(x) = ax, x < 0$

$f(x) = x, x >= 0$

O que fizemos aqui é que simplesmente substituímos a linha horizontal por uma linha não-zero, não horizontal. Aqui um é um valor pequeno como 0,01 ou algo parecido. A principal vantagem de substituir a linha horizontal é remover o gradiente zero.

Softmax

A função softmax também é um tipo de função sigmoide, mas é útil quando tentamos lidar com problemas de classificação. A função sigmoide como vimos anteriormente é capaz de lidar com apenas duas classes. O que devemos fazer quando estamos tentando lidar com várias classes? Apenas classificar sim ou não para uma única classe não ajudaria. A função softmax transforma as saídas para casa classe para valores entre 0 e 1 e também divide pela soma das saídas. Isso essencialmente dá a probabilidade de a entrada estar em uma determinada classe. Pode ser definido como:

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$

Digamos, por exemplo, que temos as saídas como [1.2, 0.9, 0.75], quando aplicamos a função softmax, obteríamos [0.42, 0.31, 0.27]. Então, agora podemos usá-los como probabilidades de que o valor seja de cada classe.

A função softmax é idealmente usada na camada de saída do classificador, onde realmente estamos tentando gerar as probabilidades para definir a classe de cada entrada.

Escolhendo a Funcão de Ativação Correta

Ufa! Muita coisa, não? E ainda não vimos as questões matemáticas envolvidas nessas funções. Mas não tenhamos pressa, não existe atalho para o aprendizado e estudaremos tudo passo a apasso, item a item, no padrão dos cursos Data Science Academy.

Agora que vimos tantas funções de ativação, precisamos de alguma lógica/heurística para saber qual função de ativação deve ser usada em qual situação. Não há uma regra de outro e a escolha depende do problema no qual você estiver trabalhando.

No entanto, dependendo das propriedades do problema, poderemos fazer uma melhor escolha para uma convergência fácil e rápida da rede neural.

- Funções Sigmóide e suas combinações geralmente funcionam melhor no caso de classificadores.
- Funções Sigmóide e Tanh às vezes são evitadas devido ao problema de Vanishing Gradient (que estudaremos no capítulo sobre redes neurais recorrentes).
- A função ReLU é uma função de ativação geral e é usada na maioria dos casos atualmente.
- Se encontrarmos um caso de neurônios deficientes em nossas redes, a função Leaky ReLU é a melhor escolha.
- Tenha sempre em mente que a função ReLU deve ser usada apenas nas camadas ocultas.
- Como regra geral, você pode começar usando a função ReLU e depois passar para outras funções de ativação no caso da ReLU não fornecer resultados ótimos.

Está começando a sentir a vibração em trabalhar com Inteligência Artificial? Então continue acompanhando, pois estamos apenas no começo! Até o próximo capitulo!

Referências:

Função Sigmóide

Machine Learning

The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition

Pattern Recognition and Machine Learning

Understanding Activation Functions in Neural Networks

Vanishing Gradient Problem

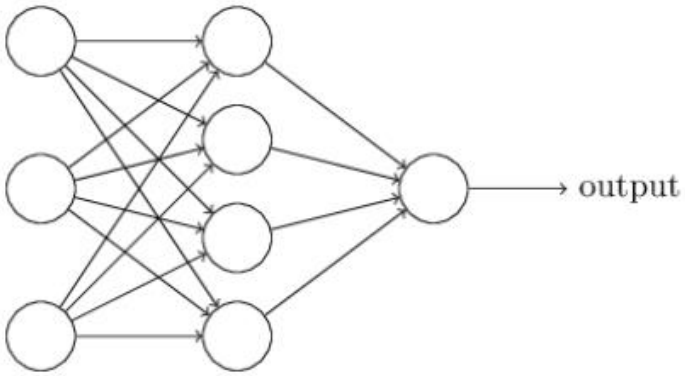
Redes Neurais, princípios e práticas

Neural Networks and Deep Learning (alguns trechos extraídos e traduzidos com autorização do autor Michael Nielsen)

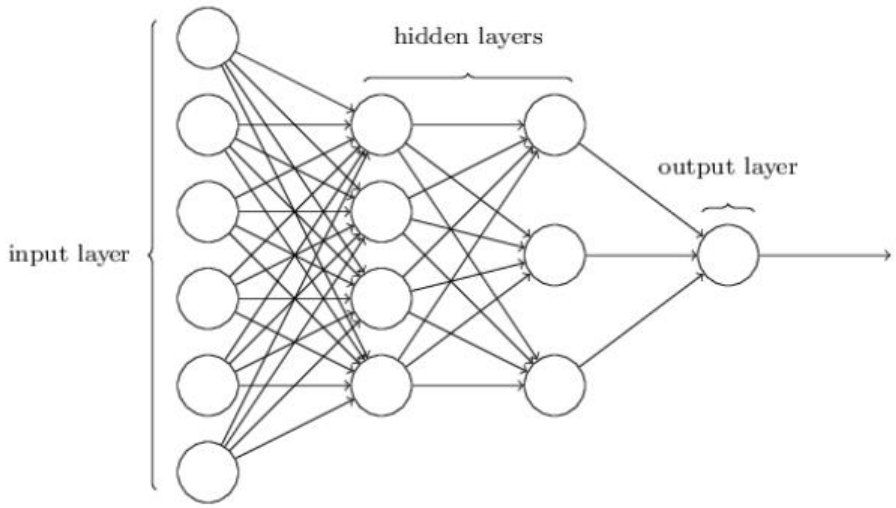
CAPÍTULO 9

A Arquitetura das Redes Neurais

No capítulo 11 vamos desenvolver uma rede neural para classificação de dígitos manuscritos, usando linguagem Python. Mas antes, vamos compreender a terminologia que será muito útil quando estivermos desenvolvendo nosso modelo, estudando a Arquitetura das redes Neurais. Suponha que tenhamos a rede abaixo:



A camada mais à esquerda nesta rede é chamada de camada de entrada e os neurônios dentro da camada soa chamados de neurônios de entrada. A camada mais à direita ou a saída contém os neurônios de saída ou, como neste caso, um único neurônio de saída. A camada do meio é chamada de camada oculta, já que os neurônios nessa camada não são entradas ou saídas. O termo "oculto" talvez soe um pouco misterioso – a primeira vez que ouvi o termo, pensei que devesse ter algum significado filosófico ou matemático profundo – mas isso realmente não significa nada mais do que "uma camada que não é entrada ou saída". A rede acima tem apenas uma única camada oculta, mas algumas redes possuem múltiplas camadas ocultas, por exemplo, a seguinte rede de quatro camadas tem duas camadas ocultas:



Tais redes de camadas múltiplas são chamadas de Perceptrons Multicamadas ou MLPs (Multilayer Perceptrons), ou seja, uma rede neural formada por Perceptrons (embora na verdade seja uma rede de neurônios sigmóides, como veremos mais adiante).

O design das camadas de entrada e saída em uma rede geralmente é direto. Por exemplo, suponha que estaos tentando determinar se uma imagem manuscrita representa um "9" ou não. Uma maneira natural de projetar a rede é codificar as intensidades dos pixels da imagem nos neurônios de entrada. Se a imagem for uma imagem em escala de cinza 64 x 64, teríamos 64 x $64 = 4.096$ neurônios de entrada, com as intensidades dimensionadas adequadamente entre 0 e 1. A camada de saída conterá apenas um único neurônio com valores inferiores a 0,5 indicando que "a imagem de entrada não é um 9" e valores maiores que 0,5 indicando que "a imagem de entrada é um 9".

Embora o design das camadas de entrada e saída de uma rede neural seja frequentemente direto, pode haver bastante variação para o design das camadas ocultas. Em particular, não é possível resumir o processo de design das camadas ocultas com poucas regras simples. Em vez disso, pesquisadores de redes neurais desenvolveram muitas heurísticas de design para as camadas ocultas, que ajudam as pessoas a obter o comportamento que querem de suas redes. Conheceremos várias heurísticas de design desse tipo mais adiante ao longo dos próximos capítulos. O design das camadas ocultas é um dos pontos cruciais em modelos de Deep Learning.

Até agora, estamos discutindo redes neurais onde a saída de uma camada é usada como entrada para a próxima camada. Essas redes são chamadas de redes neurais feedforward. Isso significa que não há loops na rede – as informações sempre são alimentadas para a frente, nunca são enviadas de volta. Se tivéssemos loops, acabaríamos com situações em que a entrada para a função σ dependeria da saída. Isso seria difícil de entender e, portanto, não permitimos tais loops.

No entanto, existem outros modelos de redes neurais artificiais em que os circuitos de feedback são possíveis. Esses modelos são chamados de redes neurais recorrentes. A ideia nestes modelos é ter neurônios que disparem por algum período de tempo limitado. Disparar pode estimular outros neurônios, que podem disparar um pouco mais tarde, também por uma duração limitada. Isso faz com que ainda mais neurônios disparem e, ao longo do tempo, conseguimos uma cascata de disparos de neurônios. Loops não causam problemas em tal modelo, uma vez que a saída de um neurônio afeta apenas sua entrada em algum momento posterior, não instantaneamente.

Geralmente, as arquiteturas de redes neurais podem ser colocadas em 3 categorias especificas:

1 – Redes Neurais Feed-Forward

Estes são o tipo mais comum de rede neural em aplicações práticas. A primeira camada é a entrada e a última camada é a saída. Se houver mais de uma camada oculta, nós as chamamos de redes neurais "profundas" (ou Deep Learning). Esses tipos de redes neurais calculam uma série de transformações que alteram as semelhanças entres os casos. As atividades dos neurônios em cada camada são uma função não-linear das atividades na camada anterior.

2 – Redes Recorrentes

Estes tipos de redes neurais têm ciclos direcionados em seu grafo de conexão. Isso significa que às vezes você pode voltar para onde você começou seguindo as setas. Eles podem ter uma dinâmica complicada e isso pode torna-los muito difíceis de treinar. Entretanto, estes tipos são mais biologicamente realistas.

Atualmente, há muito interesse em encontrar formas eficientes de treinamento de redes recorrentes. As redes neurais recorrentes são uma maneira muito natural de modelar dados sequenciais. Eles são equivalentes a redes muito profundas com uma camada oculta por fatia de tempo; exceto que eles usam os mesmos pesos em cada fatia de tempo e recebem entrada em cada fatia. Eles têm a capacidade de lembrar informações em seu estado oculto por um longo período de tempo, mas é muito difícil treiná-las para usar esse potencial.

3 – Redes Conectadas Simetricamente

Estas são como redes recorrentes, mas as conexões entre as unidades são simétricas (elas têm o mesmo peso em ambas as direções). As redes simétricas são muito mais fáceis de analisar do que as redes recorrentes. Elas também são mais restritas no que podem fazer porque obedecem a uma função de energia. As redes conectadas simetricamente sem unidades ocultas são chamadas de “Redes Hopfield”. As redes conectadas simetricamente com unidades ocultas são chamadas de “Máquinas de Boltzmann”.

Dentre estas 3 categorias, podemos listar 10 arquiteturas principais de redes neurais:

- Redes Multilayer Perceptron
- Redes Neurais Convolucionais
- Redes Neurais Recorrentes
- Long Short-Term Memory (LSTM)
- Redes de Hopfield
- Máquinas de Boltzmann
- Deep Belief Network
- Deep Auto-Encoders
- Generative Adversarial Network
- Deep Neural Network Capsules (este é um tipo completamente novo de rede neural, lançado no final de 2017)

No próximo capitulo, daremos a você uma visão geral sobre cada uma dessas 10 arquiteturas e ao longo dos capítulos seguintes, estudaremos todas elas. Cada uma dessas arquiteturas tem sido usada para resolver diferentes problemas e criar sistemas de Inteligência Artificial. Saber trabalhar com IA de forma eficiente, será determinante para seu futuro profissional.

Referências:

Formação Inteligência Artificial

Função Sigmóide

Machine Learning

The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition

Pattern Recognition and Machine Learning

Understanding Activation Functions in Neural Networks

Vanishing Gradient Problem

Redes Neurais, princípios e práticas

Neural Networks and Deep Learning (alguns trechos extraídos e traduzidos com autorização do autor Michael Nielsen)

c

O Aprendizado de Máquina (Machine Learning) é necessário para resolver tarefas que são muito complexas para os humanos. Algumas tarefas são tão complexas que é impraticável, senão impossível, que os seres humanos consigam explicar todas as nuances envolvidas. Então, em vez disso, fornecemos uma grande quantidade de dados para um algoritmo de aprendizado de máquina e deixamos que o algoritmo funcione, explorando esses dados e buscando um modelo que alcance o que os Cientistas de Dados e Engenheiros de IA determinaram como objetivo. Vejamos estes dois exemplos:

- É muito difícil escrever programas que solucionem problemas como reconhecer um objeto tridimensional a partir de um novo ponto de vista em novas condições de iluminação em uma cena desordenada. Nós não sabemos qual programa de computador escrever porque não sabemos como ocorre o processo em nosso cérebro. Mesmo se tivéssemos uma boa ideia sobre como fazê-lo, o programa poderia ser incrivelmente complicado.

- É difícil escrever um programa para calcular a probabilidade de uma transação de cartão de crédito ser fraudulenta. Pode não haver regras que sejam simples e confiáveis. Precisamos combinar um número muito grande de regras fracas. A fraude é um alvo em movimento, mas o programa precisa continuar mudando.

É onde Machine Learning pode ser aplicado com sucesso. Em vez de escrever um programa à mão para cada tarefa específica, nós coletamos muitos exemplos que especificam a saída correta para uma determinada entrada. Um algoritmo de aprendizagem de máquina recebe esses exemplos e produz um programa que faz o trabalho. O programa produzido pelo algoritmo de aprendizagem pode parecer muito diferente de um programa típico escrito à mão. Pode conter milhões de números. Se o fizermos corretamente, o programa funciona para novos casos (novos dados). Se os dados mudarem, o programa também pode mudar ao treinar em novos dados. E com a redução de custos de computação (principalmente usando processamento em nuvem), grande quantidade de dados (Big data) e processamento paralelo em GPU, temos as condições perfeitas para a evolução de Machine Learning. O maior problema, por incrível que pareça, será a falta de profissionais qualificados em número suficiente para atender as demandas do mercado.

Alguns exemplos de tarefas melhor resolvidas pela aprendizagem de máquina incluem:

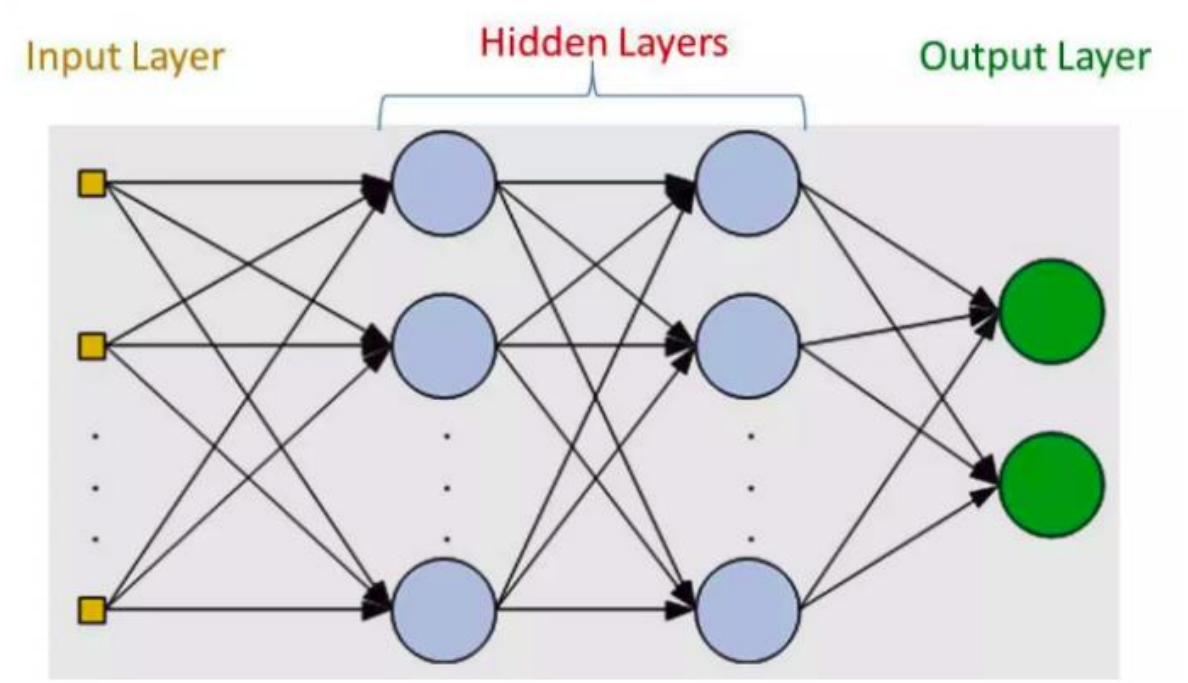
- Reconhecimento de padrões: objetos em cenas reais, identidades faciais ou expressões faciais, palavras escritas ou faladas.

- Detecção de anomalias: sequências incomuns de transações de cartão de crédito, padrões incomuns de leituras de sensores em maquinas de uma indústria têxtil.
- Previsão: preços de ações futuros ou taxas de câmbio, quais filmes uma pessoa gostaria de assistir, previsão de vendas.

Machine Learning é um campo abrangente dentro da Inteligência Artificial. Mas uma sub-área de Machine Learning, o Deep Learning (ou Redes Neurais Profundas), vem conseguindo resultados no estado da arte para as tarefas acima mencionadas. Neste capítulo você encontra as 10 Principais Arquiteturas de Redes Neurais, dentre elas as principais arquiteturas de Deep Learning.

1 – Redes Multilayer Perceptrons

O Perceptron, conforme estudamos nos capítulos anteriores, é um algoritmo simples destinado a realizar a classificação binária; isto é, prevê se a entrada pertence a uma determinada categoria de interesse ou não; fraude ou não_fraude, gato ou não_gato.



Um Perceptron é um classificador linear; ou seja, é um algoritmo que classifica a entrada separando duas categorias com uma linha reta. A entrada geralmente é um vetor de recursos **x** multiplicado por pesos **w** e adicionado a um viés (ou bias) **b.** Aqui um exemplo do Perceptron: $y = w * x + b$. Um Perceptron produz uma única saída com base em várias entradas de valor real, formando uma combinação linear usando os pesos (e às vezes passando a saída através de uma função de ativação não linear.

Rosenblatt construiu um Perceptron de uma camada. Ou seja, seu algoritmo não inclui múltiplas camadas, o que permite que as redes neurais modelem uma hierarquia de recursos. Isso impede que o Perceptron consiga realizar classificação não linear, como a função XOR (um disparador do operador XOR quando a entrada exibe uma característica ou outra, mas não ambas, significa "OR exclusivo"), como Minsky e Papert mostraram em seu livro.

Um Multilayer Perceptron (MLP) é uma rede neural artificial composta por mais de um Perceptron. Eles são compostos por uma camada de entrada para receber o sinal, uma camada de saída que toma uma decisão ou provisão sobre a entrada, e entre esses dois, um número arbitrário de camadas ocultas que são o verdadeiro mecanismo computacional do MLP, MLOs como uma camada oculta são capazes de aproximar qualquer função contínua.

O Multilayer Perceptron é uma espécie de "Hello Word" da aprendizagem profunda: uma boa forma de começar quando você está aprendendo sobre Deep Learning.

Os MLPs são frequentemente aplicados a problemas de aprendizagem supervisionados: treinam em um conjunto de pares entrada-saída e aprendem a modelar a correlação (ou dependências) entre essas entradas e saídas. O treinamento envolve o ajuste dos parâmetros, ou os pesos e bias, do modelo para minimizar o erro. O Backpropagation é usado para fazer os ajustes dos pesos e de bias em relação ao erro, e o próprio erro pode ser medido de várisa maneiras, inclusive pelo erro quadrático médio (MSE – Mean Squared Error).

As redes feed forward, como MLPs, são como ping-pong. Elas são principalmente envolvidas em dois movimentos, uma constante de ida e volta. Na passagem para frente, o fluxo de sinal se move da camada de entrada através das camadas ocultas para a camada de saída e a decisão da camada de saída é medida em relação às saídas esperadas.

Na passagem para trás, usando o Backpropagation e a regra da cadeia (Chain Rule), derivadas parciais da função de erro dos vários pesos e bias são reproduzidos através do MLP. Esse ato de diferenciação nos dá um gradiente, ao longo do qual os parâmetros podem ser ajustados à medida que movem o MLP um passo mais perto do erro mínimo. Isso pode ser feito com qualquer algoritmo de otimização baseado em gradiente, como descida estocástica do gradiente. A rede continua jogando aquele jogo de ping-pong até que o erro não possa mais ser reduzido (chegou ao mínimo possível). Este estado é conhecido como convergência.

Parece muita coisa? Sim, é. Veremos esse processo em mais detalhes aqui mesmo neste livro.

2 – Redes Neurais Convolucionais

Em 1998, Yann LeCun e seus colaboradores desenvolveram um reconhecedor, realmente bom, para dígitos manuscritos chamado LeNet. Ele usou Backpropagation em uma rede feed forward com muitas camadas ocultas, muitos mapas de unidades replicadas em cada camada, agrupando as saídas de unidades próximas, formando uma rede ampla que pode lidar com vários caracteres ao mesmo tempo, mesmo se eles se sobrepõem e uma inteligente maneira de treinar um sistema completo, não apenas um reconhecedor. Mais tarde, esta arquitetura foi formalizada sob o nome de redes convolucionais.

As Redes Neurais Convolucionais (ConvNets ou CNNs) são redes neurais artificiais profundas que podem ser usadas para classificar imagens, agrupá-las por similaridade (busca de fotos) e realizar reconhecimento de objetos dentro de cenas. São algoritmos que podem identificar rostos, indivíduos, sinais de rua, cenouras, onitorrincos e muitos outros aspectos dos dados visuais.

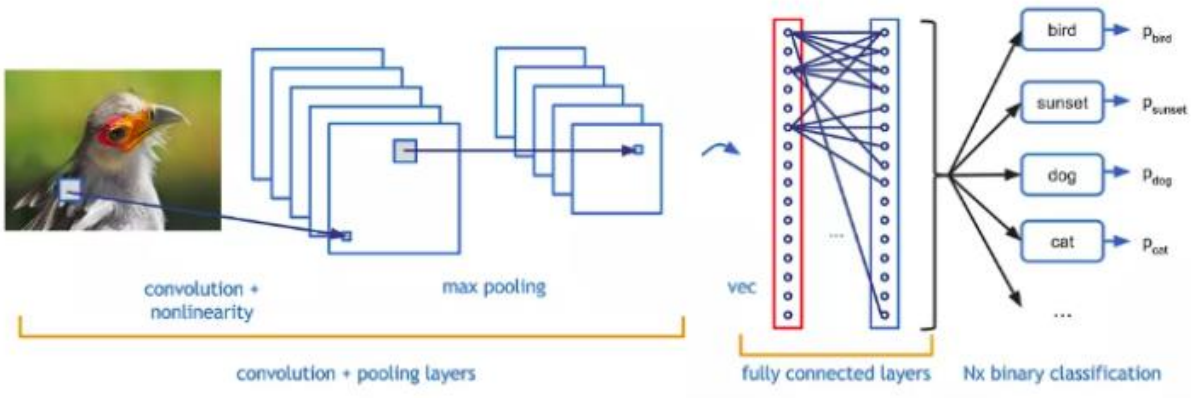
As redes Convolucionais realizam o reconhecimento óptico de caracteres (OCR) para digitalizar texto e tornar possível o processamento de linguagem natural em documentos analógicos e

manuscritos, onde as imagens são símbolos a serem transcritos. CNNs também podem ser aplicadas diretamente à análise de texto, bem como a dados gráficos.

A eficácia das redes Convolucionais no reconhecimento de imagem é uma das principais razões pelas quais o mundo testemunhou a eficácia do aprendizado profundo. Este tipo de rede está impulsionando grandes avanços em Visão Computacional, que tem aplicações óbvias em carros autônomos, robótica, drones, segurança, diagnósticos médicos e tratamentos para deficientes visuais.

As redes Convolucionais ingerem e processam imagens como tensores são matrizes de números com várias dimensões. Eles podem ser difíceis de visualizar, então vamos abordá-los por analogia. Um escalar é apenas um número, como 7; um vetor é uma lista de números (por exemplo, [7,8,9]); e uma matriz é uma grade retangular de números que ocupam várias linhas e colunas como uma planilha. Geometricamente, se um escalar é um ponto de dimensão zero, então um vetor é uma linha unidimensional. Uma matriz é um plano bidimensional, uma pilha de matrizes é um cubo tridimensional e quando cada elemento dessas matrizes tem uma pilha de matrizes é um cubo tridimensional e quando cada elemento dessas matrizes tem uma pilha de mapas de recursos ligados a ele, você entra na quarta dimensão. Clama, não se desespere (ainda). Veremos isso mais a frente com calma, quando estudarmos exclusivamente esta arquitetura. Em nossos cursos na Data Science Academy incluímos aulas completas sobre Álgebra Linear, onde escalares, vetores, matrizes e tensores são estudados na teoria e prática, pois este conhecimento é fundamental na construção de redes neurais profundas.

A primeira coisa a saber sobre redes convolucionais é que elas não percebem imagens como os humanos. Portanto, você terá que pensar de uma maneira diferente sobre o que uma imagem significa quando é alimentada e processada por uma rede convolucional.



Redes Convolucionais percebem imagens como volumes; isto é, objetos tridimensionais, em vez de estruturas planas a serem medidas apenas por largura e altura. Isso porque as imagens de cores digitais têm uma codificação vermelho-verde-azul (RGB – Red-Green-Blue), misturando essas três cores para produzir o espectro de cores que os seres humanos percebem. Uma rede convolucional recebe como três estratos separados de cores empilhadas um em cima do outro.

Assim, uma rede convolucional recebe uma imagem como uma caixa retangular cuja largura e altura são medidas pelo número de pixels ao longo dessas dimensões e cuja profundidade é de

três camadas profundas, uma para cada letra em RGB. Essas camadas de profundidade são referidas como canais.

À medida que as imagens se movem através de uma rede convolucional, descrevemos em termos de volumes de entrada e saída, expressando-as matematicamente como matrizes de múltiplas dimensões dessa forma: 30x30x3. De camada em camada, suas dimensões mudam à medida que atravessam a rede neural convolucional até gerar uma série de probabilidades na camada de saída, sendo uma probabilidade para cada possível classe de saída. Aquela com maior probabilidade, será a classe definida para a imagem de entrada, um pássaro por exemplo.

Você precisará prestar muita atenção às medidas de cada dimensão do volume da imagem, porque elas são a base das operações de álgebra linear usadas para processar imagens. Poderíamos dedicar dois capítulos inteiros a esta arquitetura. Aliás é o que faremos mais à frente no livro.

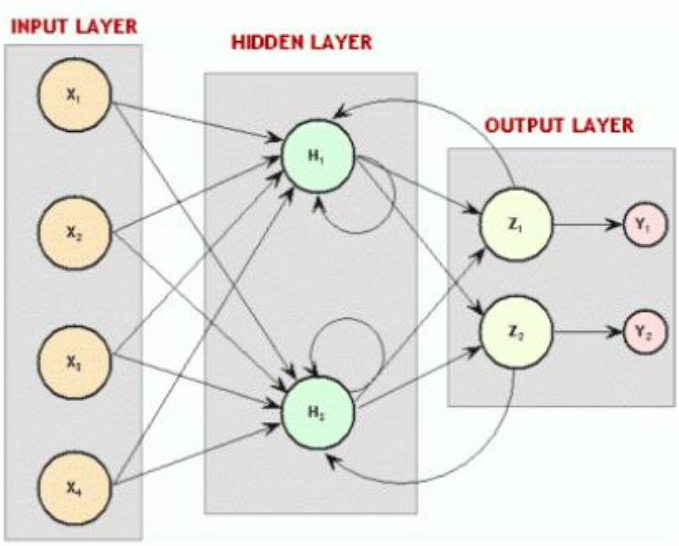
3 – Redes Neurais Recorrentes

As redes recorrentes são um poderoso conjunto de algoritmos de redes neurais artificiais especialmente úteis para o processamento de dados sequenciais, como som, dados de séries temporais ou linguagem natural. Uma versão de redes recorrentes foi usada pelo DeepMind no projeto de videogames com agentes autônomos.

As redes recorrentes diferem das redes feed forward porque incluem um loop de feedback, pelo qual a saída do passo n-1 é alimentada de volta à rede para afetar o resultado do passo n, e assim por diante para cada etapa subsequente. Por exemplo, se uma rede é exposta a uma palavra letra por letra, e é solicitado a advinhar cada letra a seguir, a primeira letra de uma palavra ajudará a determinar o que uma rede recorrente pensa que a segunda letra pode ser.

Isso difere de uma rede feed foward, que aprende a classificar cada número manuscrito por exemplo, independentemente, e de acordo com os pixels de que é exposto a partir de um único exemplo, sem se referir ao exemplo anterior para ajustar suas previsões. As redes recorrentes não enfrentam a mesma restrição um-para-um.

Embora algumas formas de dados, como imagens, não pareçam ser sequenciais, elas podem ser entendidas como sequências quando alimentadas em uma rede recorrente. Considere uma imagem de uma palavra manuscrita. Assim como as redes recorrentes processam a escrita manual, convertendo cada imagem em uma letra e usando o início de uma palavra para advinhar como essa palavra terminará, então as redes podem tratar parte de qualquer imagem como letras em uma sequência. Uma rede neural que percorre uma imagem grande pode aprender a partir de cada região, o que as regiões vizinhas, são mais prováveis de ser.



As redes recorrentes e as redes feed forward “lembram” algo sobre o mundo, modelando os dados que estão expostos. Mas elas se lembram de maneiras muito diferentes. Após o treinamento, a rede feed forward produz um modelo estático dos dados e esse modelo pode então aceitar novos exemplos e classifica-los ou agrupá-los com precisão.

Em contraste, as redes recorrentes produzem modelos dinâmicos – ou seja, modelos que mudam ao longo do tempo – de formas que produzem classificações precisas dependentes do contexto dos exemplos que estão expostos.

Para ser preciso, um modelo recorrente inclui o estado oculto que determinou a classificação anterior em uma série. Em cada etapa subsequente, esse estado oculto é combinado com os dados de entrada do novo passo para produzir a) um novo estado oculto e, em seguida, b) uma nova classificação. Cada estado oculto é reciclado para produzir seu sucessor modificado.

As memórias humanas também são conscientes do contexto, reciclando a consciência de estados anteriores para interpretar corretamente novos dados. Por exemplo, vamos considerar dois indivíduos. Um está ciente de que ele está perto da casa de Bob. O outro está ciente de que entrou em um avião. Eles interpretarão os sons “Oi Bob!” de duas formas muito diferentes, precisamos porque retém um estado oculto afetado por suas memórias de curto prazo e sensações precedentes.

Diferentes lembranças de curto prazo devem ser recontadas em momentos diferentes, a dim de atribuir o significado certo à entrada atual. Algumas dessas memórias terão sido forjadas recentemente e outras memórias terão forjado muitos passos antes de serem necessários. A rede recorrente que efetivamente associa memórias e entrada remota no tempo é chamada de Memória de Longo Prazo (LSTM), a qual veremos em seguida.

4 – Long Short-Term Memory (LSTM)

Em meados dos anos 90, a proposta dos pesquisadores alemães Sepp Hochreiter e Juergen Schmidhuber apresentou uma variação da rede recorrente com as chamadas unidades de Long

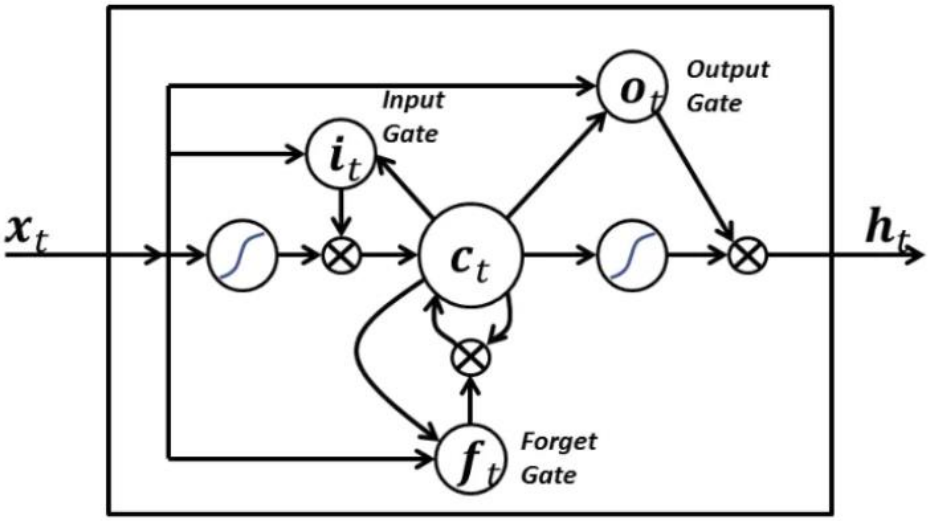
Short-Term Memory, como uma solução para o problema do Vanishing gradient, problema comum em redes neurais recorrentes.

Os LSTMs ajudam a preservar o erro que pode ser copiado por tempo e camadas. Ao manter um erro mais constante, eles permitem que as redes recorrentes continuem aprendendo durante vários passos de tempo (mais de 1000), abrindo assim um canal para vincular causas e efeitos remotamente. Este é um dos desafios centrais para a aprendizagem de máquina e a IA, uma vez que os algoritmos são frequentemente confrontados por ambientes onde os sinais de recompensa são escassos e atrasados, como a própria vida. (Os pensadores religiosos abordaram este mesmo problema com idéias de karma ou recompensas divinas, teorizando consequências invisíveis e distantes para nossas ações).

Os LSTMs contêm informações fora do fluxo normal da rede recorrente em uma célula fechada. As informações podem ser armazenadas, escritas ou lidas a partir de uma célula, como dados na memória de um computador. A célula toma decisões sobre o que armazenar, e quando permitir leituras, gravações e exclusões, através de portões abertos e fechados. Ao contrário do armazenamento digital em computadores, no entanto, esses portões são analógicos, implementados com a multiplicação de elementos por sigmóides, que estão todos na faixa de 0-1. Analógico tem a vantagem sobre o digital de ser diferenciável e, portanto, adequado para Backpropagation.

Esses portões atuam sobre os sinais que recebem e, de forma semelhante aos nós da rede neural, eles bloqueiam ou transmitem informações com base em sua força e importação, que eles filtram com seus próprios conjuntos de pesos. Esses pesos, como os pesos que modulam a entrada e estados ocultos, são ajustados através do processo de aprendizagem das redes recorrentes. Ou seja, as células aprendem quando permitirem que os dados entrem, saiam ou sejam excluídos através do processo iterativo de fazer suposições, calculando o erro durante o Backpropagation e ajustando pesos através da descida do gradiente.

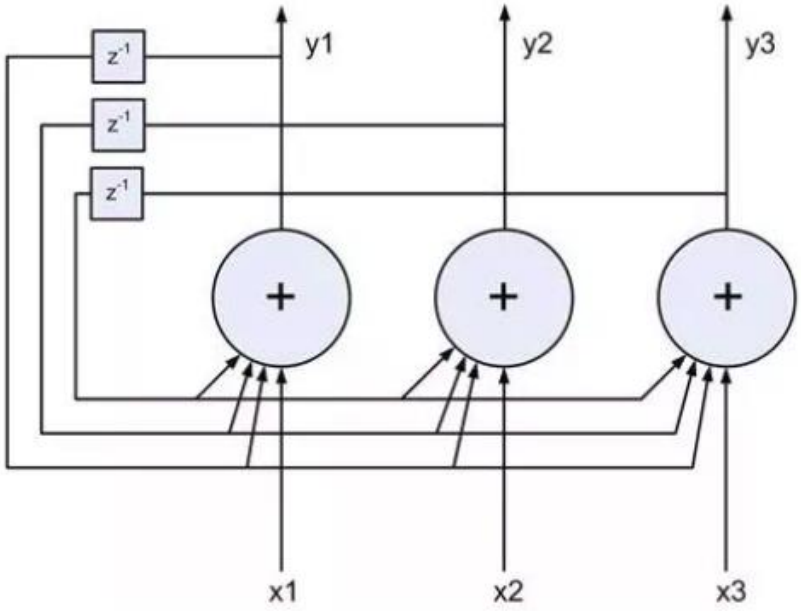
O diagrama abaixo ilustra como os dados fluem através de uma célula de memória e são controlados por seus portões.



Os LSTMs possuem muitas aplicações práticas, incluindo processamento de linguagem natural, geração automática de texto e análise de séries temporais. Teremos um capítulo inteiro dedicado aos LSTM's aqui no livro.

5 – Redes de Hopfield

Redes recorrentes de unidades não lineares geralmente são muito difíceis de analisar. Elas podem se comportar de muitas maneiras diferentes: se estabelecer em um estado estável, oscilar ou seguir trajetórias caóticas que não podem ser preditas no futuro. Uma Rede Hopfield é composta por unidades de limite binário com conexões recorrentes entre elas. Em 1982, John Hopfield percebeu que, se as conexões são simétricas, existe uma função de energia global. Cada "configuração" binária de toda a rede possui energia, enquanto a regra de decisão do limite binário faz com que a rede se conforme com um mínimo desta função de energia. Uma excelente maneira de usar esse tipo de computação é usar memórias como energia mínima para a rede neural. Usar mínimos de energia para representar memórias resulta em uma memória endereçável ao conteúdo. Um item pode ser acessado por apenas conhecer parte do seu conteúdo. É robusto contra danos no hardware.



Cada vez que memorizamos uma configuração, esperamos criar um novo mínimo de energia. Mas e se dois mínimos próximos estão em um local intermediário? Isso limita a capacidade de uma Rede Hopfield. Então, como aumentamos a capacidade de uma Rede Hopfield? Os físicos adoram a idéia de que a matemática que eles já conhecem pode explicar como o cérebro funciona. Muitos artigos foram publicados em revistas de física sobre Redes Hopfield e sua capacidade de armazenamento. Eventualmente, Elizabeth Gardner descobriu que havia uma regra de armazenamento muito melhor que usa a capacidade total dos pesos. Em vez de tentar armazenar vetores de uma só vez, ela percorreu o conjunto de treinamento muitas vezes e usou o procedimento de convergência Perceptron para treinar cada unidade para ter o estado correto, dado os estados de todas as outras unidades nesse vetor. **Os estatísticos chamam essa técnica de "pseudo-probabilidade".**

Existe outro papel computacional para as Redes Hopfield. Em vez de usar a rede para armazenar memórias, usamos para construir interpretações de entrada sensorial. A entrada é

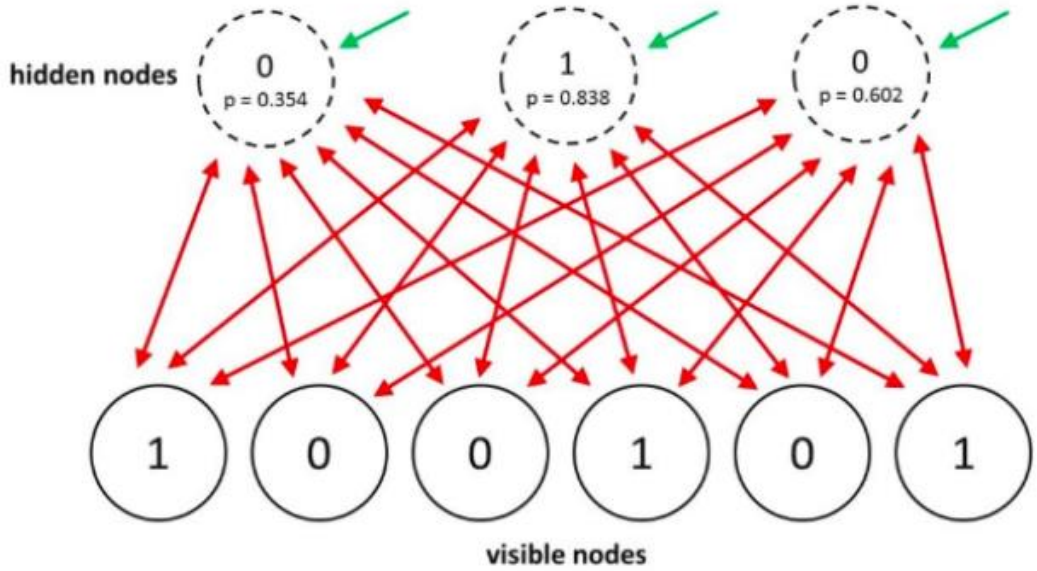
representada pelas unidades visíveis, a interpretação é representada pelos estados das unidades ocultas e o erro da interpretação é representado pela energia.

6 – Máquinas de Boltzmann

Uma Máquina de Boltzmann é um tipo de rede neural recorrente estocástica. Pode ser visto como a contrapartida estocástica e generativa das Redes Hopfield. Foi uma das primeiras redes neurais capazes de aprender representações internas e é capaz de representar e resolver problemas combinatórios difíceis.

O objetivo do aprendizado do algoritmo da Máquina de Boltzmann é maximizar o produto das probabilidades que a Máquina de Boltzmann atribui aos vetores binários no conjunto de treinamento. Isso equivale a maximizar a soma das probabilidades de log que a Máquina de Boltzmann atribui aos vetores de treinamento. Também é equivalente a maximizar a probabilidade de obtermos exatamente os N casos de treinamento se fizéssemos o seguinte: 1) Deixar a rede se estabelecer em sua distribuição estacionária no tempo N diferente, sem entrada externa e 2) Mudar o vetor visível uma vez em cada passada.

Um procedimento eficiente de aprendizado de mini-lote foi proposto para as Máquinas de Boltzmann por Salakhutdinov e Hinton em 2002.



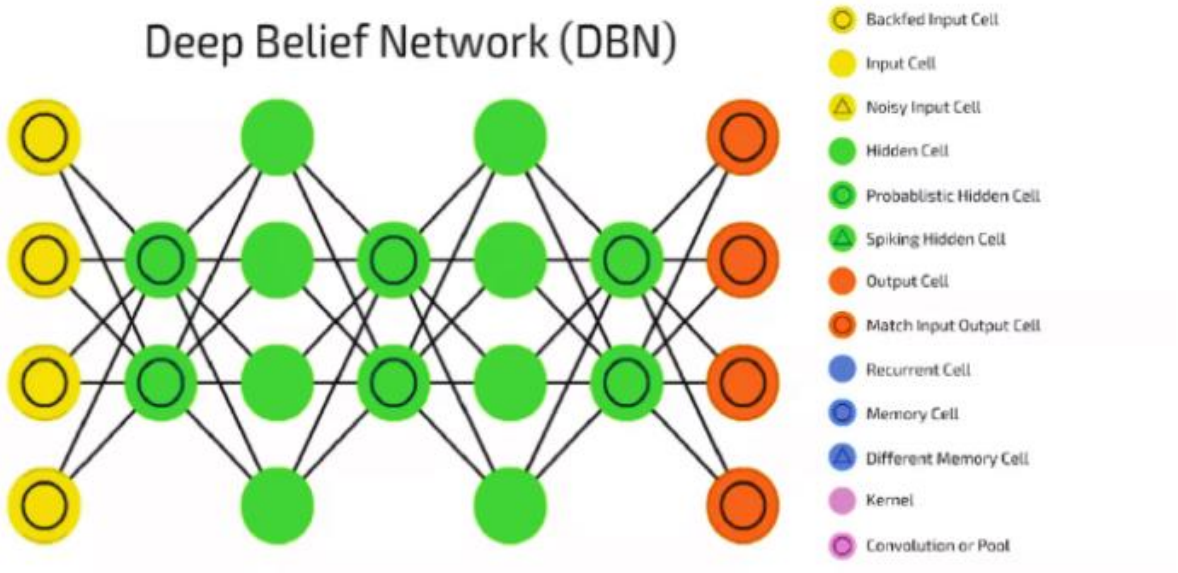
Em uma Máquina de Boltzmann geral, as atualizações estocásticas de unidades precisam ser sequenciais. Existe uma arquitetura especial que permite alternar atualizações paralelas que são muito mais eficientes (sem conexões dentro de uma camada, sem conexões de camada ignorada). Este procedimento de mini-lote torna as atualizações da Máquina de Boltzmann mais paralelas. Isso é chamado de Deep Boltzmann Machines (DBM), uma Máquina de Boltzmann geral, mas com muitas conexões ausentes.

Em 2014, Salakhutdinov e Hinton apresentaram outra atualização para seu modelo, chamando-o de Máquinas de Boltzmann Restritas. Elas restringem a conectividade para facilitar a interferência e a aprendizagem (apenas uma camada de unidades escondidas e sem conexões entre unidades ocultas). Em um RBM, é preciso apenas um passo para alcançar o equilíbrio.

7 – Deep Belief Network

O Backpropagation é considerado o método padrão em redes neurais artificiais para calcular a contribuição de erro de cada neurônio após processar um lote de dados (teremos um capítulo inteiro sobre isso). No entanto, existem alguns problemas importantes no Backpropagation. Em primeiro lugar, requer dados de treinamento rotulados; enquanto quase todos os dados estão sem rótulos. Em segundo lugar, o tempo de aprendizagem não escala bem, o que significa que é muito lento em redes com múltiplas camadas ocultas. Em terceiro lugar, pode ficar preso em um "local optima". Portanto, pare redes profundas, o backpropagation está longe de ser ótimo.

Para superar as limitações do backpropagation, os pesquisadores consideram o uso de abordagens de aprendizado sem supervisão. Isso ajuda a manter a eficiência e a simplicidade de usar um método de gradiente para ajustar os pesos, mas também usá-lo para modelar a estrutura da entrada sensorial. Em particular, eles ajustam os pesos para maximizar a probabilidade de um modelo ter gerado a entrada sensorial. A questão é que tipo de modelo generativo devemos aprender? Pode ser um modelo baseado em energia como uma Máquina de Boltzmann? Ou um modelo causal feito de neurônios? Ou um híbrido dos dois?



Uma Deep Belief Network pode ser definida como uma pilha de Máquinas de Boltzmann Restritas (RBM – Restricted Boltzmann Machines), em que cada camada RBM se comunica com as camadas anterior e posterior. Os nós de qualquer camada não se comunicam lateralmente.

Esta pilha de RBMs pode terminar com uma camada Softmax para criar um classificador, ou simplesmente pode ajudar a agrupar dados não gravados em um cenário de aprendizado sem supervisão.

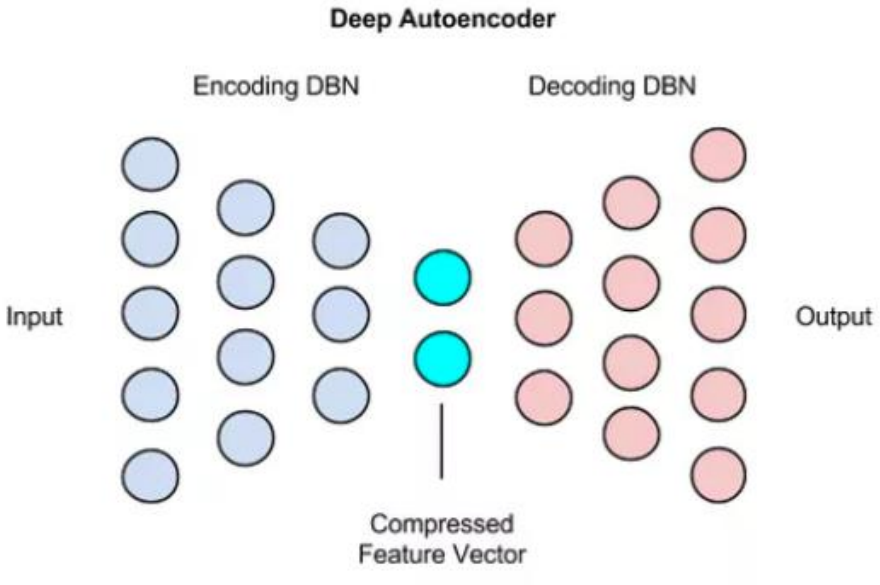
Com a exceção das camadas inicial e final, cada camada em uma Deep Belief Network tem uma função dupla: ela serve como a camada oculta para os nós que vem antes, e como a camada de entrada (ou "visível") para os nós que vem depois. É uma rede construída de redes de camada única.

As Deep Belief Networks são usadas para reconhecer, agrupar e gerar imagens, sequencias de vídeos e dados de captura de movimento. Outra aplicação das Deep Belief Networks é no Processamento de Linguagem Natural. Esse tipo de rede foi apresentado por Geoff Hinton e seus alunos em 2006.

8 – Deep Auto-Encoders

Um Deep Auto-Encoder é composto por duas redes simétricas Deep Belief que tipicamente têm quatro ou cinco camadas rasas que representam a metade da codificação (encoder) da rede e o segundo conjunto de quatro ou cinco camadas que compões a metade da decodificação (decoder).

As camadas são Máquinas de Boltzmann Restritas, os blocos de construção das Deep Belief Networks, com várias peculiaridades que discutiremos abaixo. Aqui está um esquema simplificado da estrutura de um Deep Auto-Encoder:



Os Deep Auto-Encoders são uma maneira muito agradável de reduzir a dimensionalidade não linear devido a alguns motivos: eles fornecem mapeamentos flexíveis em ambos os sentidos. O tempo de aprendizagem é linear (ou melhor) no número de casos de treinamento. E o modelo de codificação final é bastante compacto e rápido. No entanto, pode ser muito difícil otimizar

Deep Auto-Encoders usando backpropagation. Com pequenos pesos iniciais, o gradiente do backpropagation morre. Mas temos maneiras de otimizá-las, usando o pré-treinamento camada-por-camada sem supervisão ou apenas inicializando os pesos com cuidado.

Os Deep Auto-Encoders são úteis na modelagem de tópicos ou modelagem estatística de tópicos abstratos que são distribuídos em uma coleção de documentos, Isso, por sua vez, é um passo importante em sistemas de perguntas e respostas como o IBM Watson.

Em resumo, cada documento em uma coleção é convertido em um Bag-of-Words (ou seja, um conjunto de contagens de palavras) e essas contagens de palavras são dimensionadas para decimais entre 0 e 1, o que pode ser pensado como a probabilidade de uma palavra ocorrer no documento.

As contagens de palavras em escala são então alimentadas em uma Deep Belief Network, uma pilha de Máquinas de Boltzmann Restritas, que elas mesmas são apenas um subconjunto de Auto-Encoders. Essas Deep Belief networks, ou DBNs, comprimem cada documento para um conjunto de 10 números através de uma série de transformações sigmóides que o mapeiam no espaço de recursos.

O conjunto de números de cada documento, ou vetor, é então introduzido no mesmo espaço vetorial, e sua distância de qualquer outro vetor de documento medido. Em termos aproximados, os vetores de documentos próximos se enquadram no mesmo tópico. Por exemplo, um documento poderia ser a "pergunta" e outros poderiam ser as "respostas", uma combinação que o software faria usando medidas de espaço vetorial.

Em resumo, existem agora muitas maneiras diferentes de fazer pré-treinamento camada-por-camada de recursos. Para conjuntos de dados que não possuem um grande número de casos rotulados, o pré-treinamento ajuda a aprendizagem discriminativa subsequente. Para conjuntos de dados muito grandes e rotulados, não é necessário iniciar os pesos utilizados na aprendizagem supervisionada usando pré-treinamento não supervisionado, mesmo para redes profundas. O pré-treinamento foi o primeiro bom caminho para inicializar os pesos para redes profundas, mas agora existem outras formas. Mas se construirmos redes muito maiores, precisaremos de pré-treinamento novamente!

9 – Generative Adversarial Network

As Generative Adversarial Networks (GANs) são arquiteturas de redes neurais profundas compostas por duas redes, colocando uma contra a outra (daí o nome "adversária").

Os GANs foram introduzidos em um artigo de Ian Goodfellow e outros pesquisadores da Universidade de Montreal no Canadá, incluindo Yoshua Bengio, em 2014. Referindo-se aos GANs, o diretor de pesquisa da IA do Facebook, Yann LeCun, chamou de treinamento adversário "a idéia mais interessante nos últimos 10 anos em Machine Learning".

O potencial de GANs é enorme, porque eles podem aprender a imitar qualquer distribuição de dados. Ou seja, os GANs podem ser ensinados a criar mundos estranhamente semelhantes aos nossos em qualquer domínio: imagens, música, fala, prosa. Eles são artistas robôs em um sentido, e sua produção é impressionante – até mesmo pungente.

Para entender os GANs, você deve saber como os algoritmos geradores funcionam, e para isso, contrastá-los com algoritmos discriminatórios é útil. Os algoritmos discriminatórios tentam classificar dados de entrada; isto é, dados os recursos de uma instancia de dados, eles predizem um rótulo ou categoria a que esses dados pertencem.

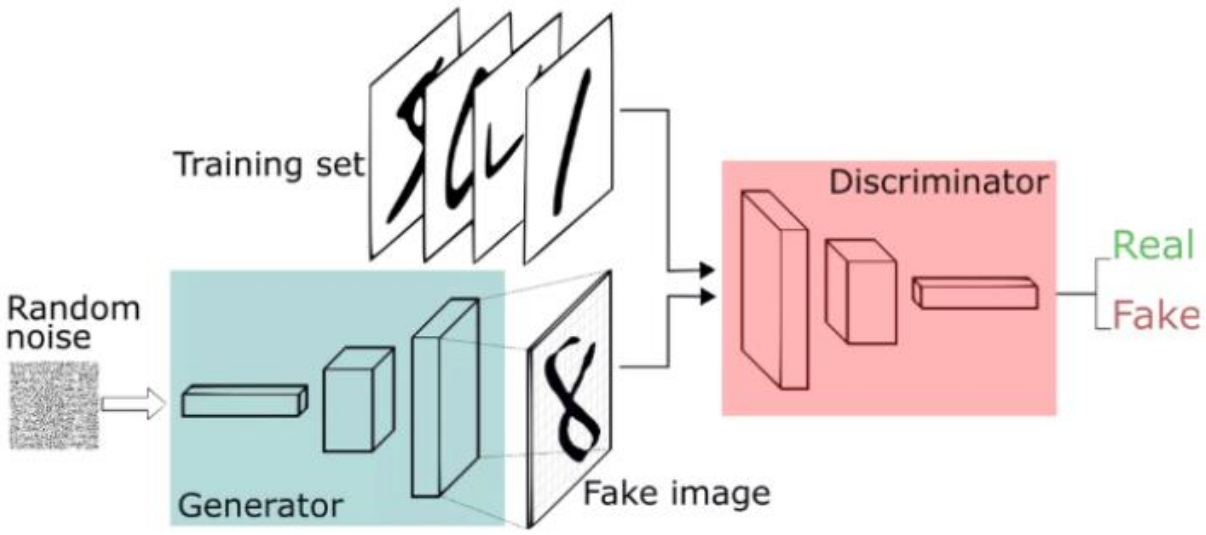
Por exemplo, tendo em conta as palavras em um e-mail, um algoritmo discriminatório pode prever se a mensagem é spam ou not_spam. O spam é um dos rótulos e o saco de palavras (Bag of Words) coletados do e-mail são os recursos que constituem os dados de entrada. Quando est problema é expresso matematicamente, o rótulo é chamado y e os recursos são chamados de x. A formulação p (y | x) é usada para significar "a probabilidade de y dado x", que neste caso seria traduzido para "a probabilidade de um email ser spam e y dado x", que neste caso seria traduzido para "a probabilidade de um email ser spam com as palavras que contém".

Portanto, algoritmos discriminatórios mapeiam recursos para rótulos. Eles estão preocupados apenas com essa correlação. Uma maneira de pensar sobre algoritmos generativos é que eles fazem o contrário. Em vez de prever um rótulo com determinados recursos, eles tentam prever os recursos com um determinado rótulo.

A questão que um algoritmo gerador tenta responder é: assumir que este email é spam, qual a probabilidade dos recursos? Enquanto os modelos discriminatórios se preocupam mais com a relação entre y e x, os modelos generativos se preocupam com "como você obtêm x". Eles permitem que você capture p (x | y), a probabilidade de x dado y, ou a probabilidade de características oferecidas em uma classe. (Dito isto, os algoritmos geradores também podem ser usados como classificadores, embora eles podem fazer mais do que categorizar dados de entrada.)

Outra maneira de pensar sobre isso é distinguir discriminativo de gerador assim:

- Modelos discriminativos aprendem o limite entre as classes
- Modelos generativos modelam a distribuição de classes individuais



Uma rede neural, chamada de gerador, gera novas instâncias de dados, enquanto a outra, o discriminador, as avalia por autenticidade; ou seja, o discriminador decide se cada instancia de dados que revisa pertence ao conjunto de dados de treinamento real ou não.

Digamos que estamos tentando fazer algo mais banal do que imitar a Mona Lisa. Vamos gerar números escritos à mão como os encontrados no conjunto de dados MNIST, que é retirado do mundo real. O objeto do discriminador, quando mostrado uma instância do verdadeiro conjunto de dados MNIST, é reconhece-los como autênticos.

Enquanto isso, o gerador está criando novas imagens que passa para o discriminador. Sso acontece com a esperança de que eles, também, sejam considerados autênticos, embora sejam falsos. O objetivo do gerador é gera dígitos escritos por si mesmo. O objetivo do discriminador é identificar as imagens provenientes do gerador como falsas.

Aqui estão os passos que um GAN realiza:

- O gerador recebe números aleatórios e retorna uma imagem.
- Essa imagem gerada é alimentada no discriminador ao lado de um fluxo de imagens tirada do conjunto de dados real.
- O discriminador assume imagens reais e falsas e retorna probabilidades, um número entre 0 e 1, com 1 representando uma previsão de autenticidade e 0 representando falsas.

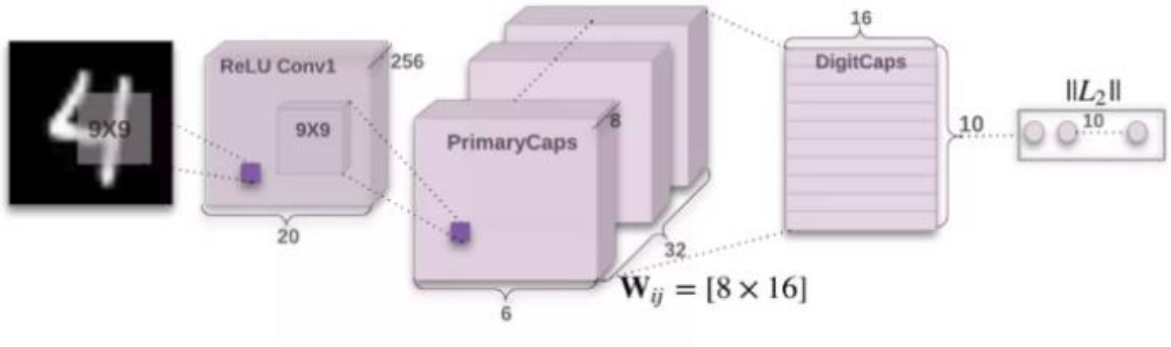
Então você tem um loop de feedback duplo:

- Um discriminador está em um loop de feedback com as imagens verdadeiras, que conhecemos.
- O gerador está em um loop de feedback com o discriminador.

10 – Deep Neural Network Capsules

No final de 2017, Geoffrey Hinton e sua equipe publicaram dois artigos que introduziram um novo tipo de rede chamada ***Capsules***. Além disso, a equipe publicou um algoritmo, denominado roteamento dinâmico entre cápsulas, que permite treinar essa rede.

Para todos na comunidade de Deep Learning, esta é uma grande notícia, e por várias razões. Em primeiro lugar, Hinton é m dos fundadores do Deep Learning e um inventor de inúmeros modelos e algoritmos que hoje são amplamente utilizados. Em segundo lugar, esses artigos apresentam algo completamente novo, e isso é muito emocionante porque provavelmente estimulará a onda adicional e aplicativos muito inovadores.



As ***Capsules*** introduzem um novo bloco de construção que pode ser usado na aprendizagem profunda para modelar melhor as relações hierárquicas dentro da representação do conhecimento interno de uma rede neural, A intuição por trás deles é muito simples e elegante

Hinton e sua equipe propuseram uma maneira de treinar essa rede composta de cápsulas e treinou-a com êxito em um conjunto de dados simples, alcançando desempenho de ponta. Isso é muito encorajador. No entanto, há desafios. As implementações atuais são muito mais lentas do que outros modelos modernos de aprendizado profundo. O tempo mostrará se as redes ***Capsules*** podem ser treinadas de forma rápida e eficiente. Além disso, precisamos ver se elas funcionam bem em conjuntos de dados mais difíceis e me diferentes domínios.

Em qualquer caso, a rede ***Capsule*** é um modelo muito interessante e já funcionando, que definitivamente se desenvolverá ao longo do tempo e contribuirá para uma maior expansão de aplicações de aprendizagem profunda.

Incluímos as Capsules entre as 10 principais arquiteturas de redes neurais, pois elas representam a inovação e o avanço na incrível e vibrante área de Deep Learning e sistemas de Inteligência Artificial. Profissionais que realmente desejem abraçar a IA como carreira, devem estar atentos aos movimentos e inovações na área.

Esta não é uma lista definitiva de arquiteturas e existem outras, tais como Word2Vex, Doc2vec, Neural Embeddings e variações das arquiteturas aqui apresentadas, como Denoising Autoencoders, Variational Autoencoders, além de outras categorias como Deep Reinforcement Learning. Exatamente para auxiliar aqueles que buscam conhecimento de ponta 100% em português e 100% online, que nós criamos a Formação Inteligência Artificial, o único programa do Brasil completo, com todas as ferramentas que o aluno precisa para aprender a trabalhar com IA de forma eficiente. O aluno aprende programação paralela em GPU, Deep Learning e seus frameworks, estuda as principais arquiteturas com aplicações práticas e desenvolve aplicações de Visão computacional e Processamento de Linguagem Natural.

Isso conclui a primeira parte deste livro, com uma introdução ao universo do Deep Learning. No próximo capítulo começaremos a ver as redes neurais em ação. Até lá.

Referências:

Formação Inteligência Artificial

Machine Learning

The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition

Pattern Recognition and Machine Learning

Understanding Activation Functions in Neural Networks

Multilayer Perceptrons

Long Short-Term Memory

Long Short-Term Memory Neural Computation

The Unreasonable Effectiveness of Recurrent Neural Networks

Perceptrons, Expanded Edition

The 8 Neural Network Architectures Machine Learning Researchers Need to Learn

Dynamic Routing Between Capsules

Neural networks and physical systems with emergent collective computational abilities

Deep Boltzmann Machines

Application of Deep Belief Networks for Natural Language Understanding

A fast learning algorithm for deep belief nets

A Beginner's Guide to Deep Autoencoders

Matrix Capsules With RM Routing

Generative Adversarial Networks (GANs)

CAPÍTULO 11

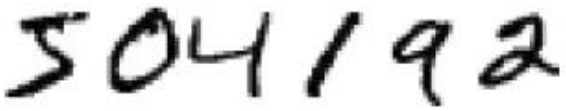
Design de uma Rede Neural para Reconhecimento de Dígitos

Na primeira parte deste livro online, durante os 10 primeiros capítulos definimos e estudamos o universo das redes neurais artificias. Neste ponto você já deve ter uma boa compreensão sobre que são estes algoritmos e como podem ser usados, além da importância das redes neurais para a construção de sistemas de Inteligência Artificial. Estamos prontos para iniciar a construção de redes neurais e na sequência estudaremos as arquiteturas mais avançadas. Vamos começar definindo o Design De Uma Rede Neural Para Reconhecimento de Digitos,

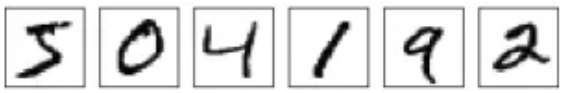
Nossa primeira tarefa será construir uma rede neural para reconhecer caligrafia, ou seja, digitos escritos à mão que foram digitalizados em imagens no computador. Por que vamos começar com este tipo de tarefa? Porque ela permite percorrer todas as etapas e procedimentos matemáticos de uma rede neural, sendo, portanto, uma excelente introdução. Vamos começar?

Se você acompanha os cursos na Data Science Academy já sabe que: antes de pensar em escrever sua primeira linha de código, é preciso definir claramente o problema a ser resolvido. A tecnologia existe para resolver problemas e a definição clara do objetivo é o ponto de partida de qualquer projeto de sucesso! Neste capítulo definiremos o problema a ser resolvido, nesse caso o reconhecimento de dígitos manuscritos.

Podemos dividir o problema de reconhecer os dígitos manuscritos em dois sub-problemas. Primeiro, precisamos encontrar uma maneira de quebrar uma imagem que contenha muitos dígitos em uma sequência de imagens separadas, cada uma contendo um único dígito. Por exemplo, gostaríamos de quebrar a imagem:



em seis imagens separadas:

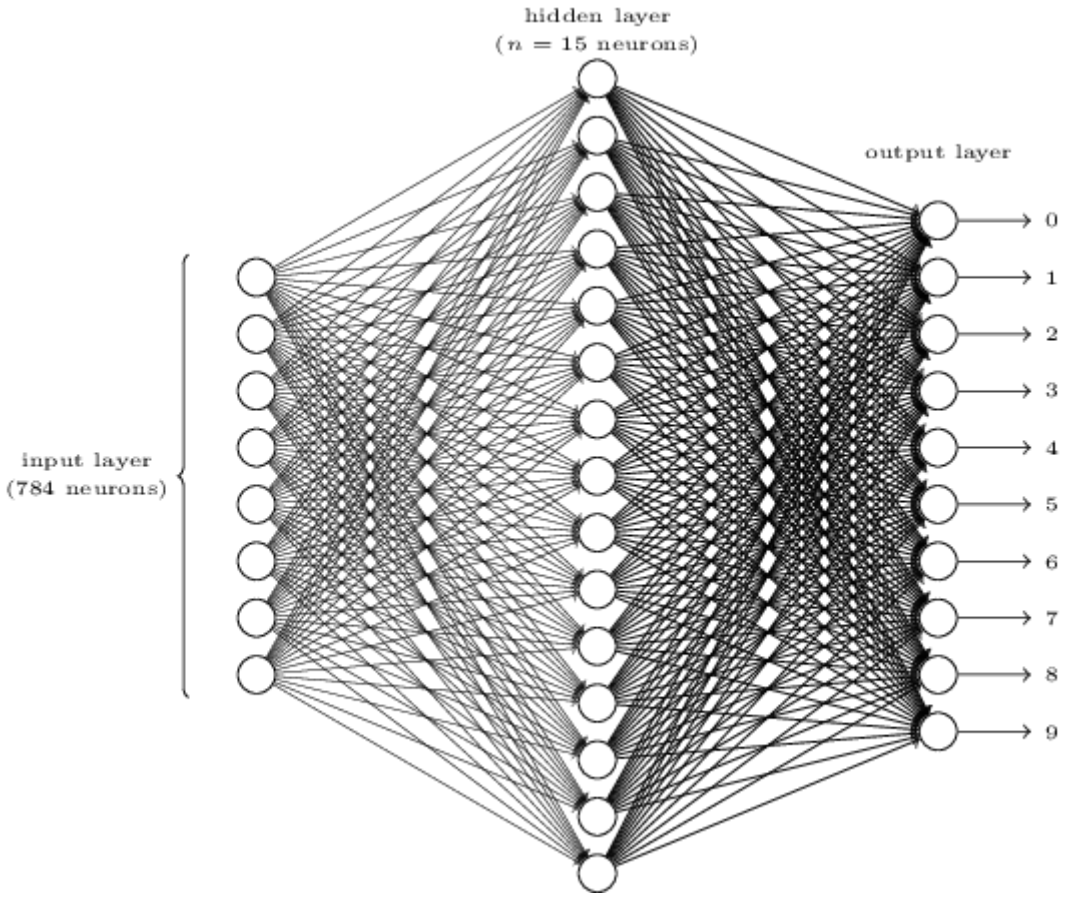


Nós humanos resolvemos esse problema de segmentação com facilidade, mas é um desafio para um programa de computador dividir corretamente a imagem. Uma vez que a imagem foi segmentada, o programa precisa classificar cada dígito individual. Então, por exemplo,

gostaríamos que nosso programa reconhecesse automaticamente que o primeiro dígito acima é um 5:

Vamos nos concentrar em escrever um programa para resolver o segundo problema, isto é, classificar dígitos individuais. O problema da segmentação não é tão difícil de resolver uma vez que você tenha uma boa maneira de classificar os dígitos individuais. Existem muitas aborda abordagens para resolver o problema de segmentação. Uma abordagem é testar muitas maneiras diferentes de segmentar a imagem, usando o classificador de digitas individuais para marcar cada segmentação de teste.Uma segmentação de teste obtém uma pontuação alta se o classificador de dígitos individuais estiver confiante de sua classificação em todos os segmentos e uma pontuação baixa se o classificador tiver muitos problemas em um ou mais segmentos. A idéia é que, se o classificador estiver tendo problemas em algum lugar, provavelmente está tendo problemas porque a segmentação foi escolhida incorretamente. Essa idéia e outras variações podem ser usadas para resolver o problema de segmentação. Então, em vez de se preocupar com a segmentação, nos concentramos no desenvolvimento de uma rede neural que pode resolver o problema mais interessante e difícil, ou seja, reconhecer dígitos individuais manuscritos.

Para reconhecer dígitos individuais, usaremos uma rede neural de três camadas:



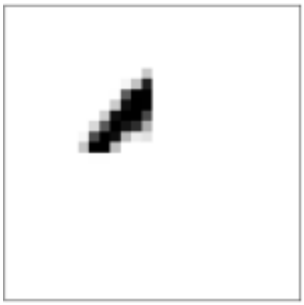
A camada de entrada da rede contém neurônios que codificam os valores dos pixels de entrada. Conforme iremos discutir no próximo capítulo, nossos dados de treinamento para a rede consistirão em muitas imagens de 28 por 28 pixels de dígitos manuscritos digitalizados e, portanto, a camada de entrada contém $28 \times 28 = 784$ neurônios (Nota: uma imagem nada mais é do que uma matriz, nesse caso de dimensões 28×28, que iremos converter em um vetor cujo tamanho será 784, onde cada item representa um pixel na imagem). Os pixels de entrada são de escala de cinza, com um valor de 0.0 representando branco e um valor de 1.0 representando preto. Valores intermediários representam tonalidades gradualmente escurecidas de cinza.

A segunda camada da rede é uma camada oculta. Representaremos o número de neurônios nesta camada oculta por n, e vamos experimentar diferentes valores para n. O exemplo mostrado acima ilustra uma pequena camada oculta, contendo apenas $n = 15$ neurônios.

A camada de saída da rede contém 10 neurônios. Se o primeiro neurônio "disparar" (for ativado), ou seja, tiver uma saída ≈ 1, então isso indicará que a rede acha que o dígito é 0. Se o segundo neurônio "disparar" (for ativado), isso indicará que a rede pensa que o dígito é um 1. E assim por diante. Em resumo, vamos numerar os neurônios de saída de 0 a 9 e descobrimos qual neurônio possui o maior valor de ativação. Se esse neurônio é, digamos, neurônio número 6, então nossa rede adivinhará que o dígito de entrada era um 6. E assim por diante para os outros neurônios de saída.

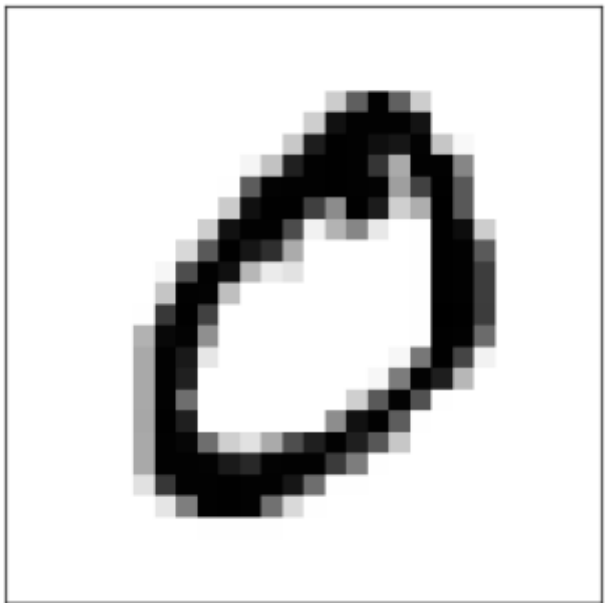
Você pode se perguntar por que usamos 10 neurônios de saída. Afinal, o objetivo da rede é nos dizer qual dígito (0,1,2, …, 9) corresponde à imagem de entrada. Uma maneira aparentemente natural de fazer isso é usar apenas 4 neurônios de saída, tratando cada neurônio como assumindo um valor binário, dependendo se a saída do neurônio está mais próxima de 0 ou 1. Quatro neurônios são suficientes para codificar a resposta, desde que $2\hat{}4 = 16$ é mais do que os 10 valores possíveis para o dígito de entrada. Por que nossa rede deve usar 10 neurônios em vez disso? Isso não é ineficiente? A justificativa final é empírica: podemos experimentar ambos os projetos de rede, e verifica-se que, para este problema específico, a rede com 10 neurônios de saída aprende a reconhecer dígitos melhor do que a rede com 4 neurônios de saída. Mas isso ainda deixa a pergunta por que o uso de 10 neurônios de saída funciona melhor. Existe alguma heurística que nos diga com antecedência que devemos usar a codificação de 10 saídas em vez da codificação de 4 saídas?

Entender porque fazemos isso, ajuda a pensar sobre o que a rede neural está realmente fazendo. Considere primeiro o caso em que usamos 10 neurônios de saída. Vamos nos concentrar no primeiro neurônio de saída, aquele que está tentando decidir se o dígito é ou não 0. Ele faz isso pesando evidências da camada oculta dos neurônios. O que esses neurônios ocultos estão fazendo? Bem, vamos supor que o primeiro neurônio na camada oculta detecta ou não uma imagem como a seguinte:



Isso pode ser feito pesando fortemente pixels de entrada que se sobrepõem à imagem e apenas ponderam ligeiramente as outras entradas. De forma semelhante, suponhamos que o segundo, terceiro e quarto neurônios na camada oculta detectem se as seguintes imagens estão ou não presentes:

Como você pode ter adivinhado, essas quatro imagens juntas compõem a imagem 0 que vimos na linha de dígitos mostrada anteriormente:



Então, se todos os quatro neurônios ocultos estão disparando, podemos concluir que o dígito é um 0. Claro, esse não é o único tipo de evidência que podemos usar para concluir que a imagem era um 0 – podemos legitimamente obter um 0 em muitas outras maneiras (por exemplo, através de traduções das imagens acima, ou pequenas distorções). Mas parece seguro dizer que, pelo menos neste caso, concluiríamos que a entrada era um 0.

Supondo que a rede neural funciona assim, podemos dar uma explicação plausível sobre porque é melhor ter 10 saídas da rede, em vez de 4. Se tivéssemos 4 saídas, o primeiro neurônio de saída tentaria decidir o que mais um bit significativo do dígito representa. E não existe uma maneira fácil de relacionar esse bit mais significativo com formas simples, como as mostradas acima. As formas componentes do dígito estarão intimamente relacionadas com (digamos) o bit mais significativo na saída.

Isso tudo é apenas uma heurística. Nada diz que a rede neural de três camadas tem que operar da maneira que descrevemos, com os neurônios ocultos detectando formas de componentes simples. Talvez um algoritmo de aprendizado inteligente encontre alguma atribuição de pesos que nos permita usar apenas 4 neurônios de saída. Mas, usar uma boa heurística pode economizar muito tempo na concepção de boas arquiteturas de redes neurais.

Já temos então um design para a nossa rede neural. Agora precisamos definir como será o processo de aprendizagem do algoritmo, antes de começar a codificar nossa rede em linguagem Python. Usaremos o treinamento com Gradiente Descendente, assunto do próximo capítulo, que aliás eu não perderia por nada, se fosse você, pois aí está a “magia” por trás das redes neurais. Até lá!

Para acompanhar os próximos capítulos e reproduzir os exemplos, você deve ter o Anaconda Python instalado no seu computador com Python versão 3.6.x. Acesse o capítulo 1 do curso gratuito Python Fundamentos Para Análise de Dados, para aprender como instalar o Anaconda.

Referências:

Formação Inteligência Artificial

Função Sigmóide

Machine Learning

The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition

Pattern Recognition and Machine Learning

Understanding Activation Functions in Neural Networks

Redes Neurais, princípios e práticas

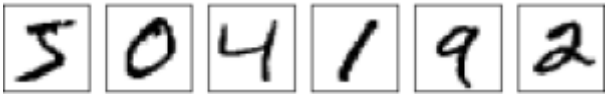
Neural Networks and Deep Learning (alguns trechos extraídos e traduzidos com autorização do autor Michael Nielsen)

CAPÍTULO 12

Aprendizado com a Descida do Gradiente

No capítulo anterior definimos o design para a nossa rede neural e agora podemos começar o processo de aprendizado de máquina. Neste capítulo você vai compreender o que é o Aprendizado Com a Descida do Gradiente.

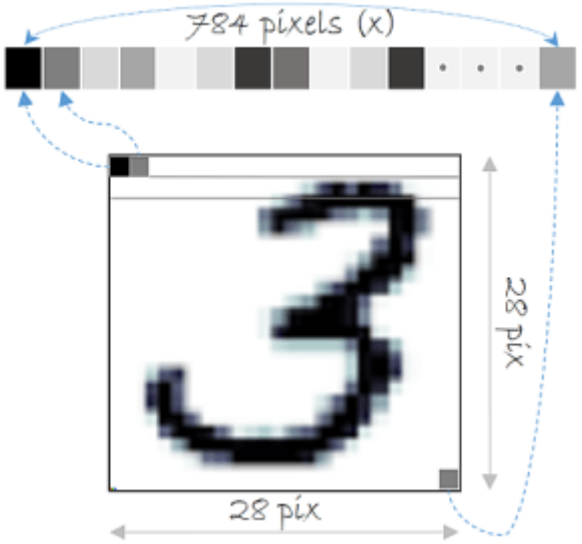
A primeira coisa que precisamos é um conjunto de dados para o treinamento da rede. Usaremos o conjunto de dados MNIST, que contém dezenas de milhares de imagens digitalizadas de dígitos manuscritos, juntamente com suas classificações corretas. O nome MNIST vem do fato de que é um subconjunto modificado de dois conjuntos de dados coletados pelo NIST, o Instituto Nacional de Padrões e Tecnologia dos Estados Unidos. Aqui estão algumas imagens do MNIST:



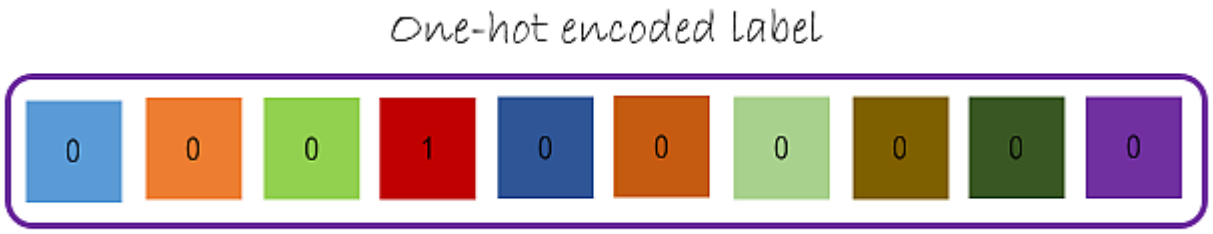
O MNIST tem duas partes. A primeira parte contém 60.000 imagens para serem usadas como dados de treinamento. Essas imagens são amostras de manuscritos escaneados de 250 pessoas, metade dos quais funcionários do Bureau do Censo dos EUA e metade dos estudantes do ensino médio. As imagens estão em escala de cinza e 28 por 28 pixels de tamanho. A segunda parte do conjunto de dados MNIST tem 10.000 imagens a serem usadas como dados de teste, também 28 por 28 pixels em escala de cinza. Usaremos os dados do teste para avaliar o quão bem a nossa

rede neural aprendeu a reconhecer os dígitos. Para fazer deste um bom teste de desempenho, os dados de teste foram retirados de um conjunto diferente de 250 pessoas em relação aos dados de treinamento originais (embora ainda seja um grupo dividido entre funcionários do Census Bureau e alunos do ensino médio). Isso nos ajuda a confiar que nosso sistema pode reconhecer dígitos de pessoas cuja escrita não viu durante o treinamento.

Usaremos a notação x para indicar uma entrada (input) de treinamento. Será conveniente considerar cada entrada de treinamento x (cada imagem) como um vetor de 784 posições (28 x 28 pixels). A imagem abaixo representa como este vetor é construído:



Cada entrada no vetor representa o valor de cinza para um único pixel na imagem. Vamos indicar a saída correspondente desejada por y = y(x), onde y é um vetor com dimensão 10. Por exemplo, se uma imagem de treinamento particular, x, representa um 3, então y(x) = (0,0,0,1,0,0,0,0,0,0)T é a saída desejada da rede . Observe que T aqui é a operação de transposição, transformando um vetor de linha em um vetor comum (coluna). Vamos deixar isso mais claro. Observe a figura abaixo:



Vamos usar os pixels de imagem correspondentes ao fluxo inteiro chamado “features”. Os rótulos são One-Hot Encoded 1-hot. O rótulo que representa a classe de saída da imagem com dígito 3 torna-se “0001000000” uma vez que temos 10 classes para os 10 dígitos possíveis, onde o primeiro índice corresponde ao dígito “0” e o último corresponde ao dígito “9”.

O que queremos é um algoritmo que nos permita encontrar pesos e bias para que a saída da rede se aproxime de y(x) para todas as entradas de treinamento x. Para quantificar o quão bem estamos alcançando esse objetivo, definimos uma função de custo:

$$C(w,b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2$$

Função de Custo Quadrático

Na fórmula acima, w indica a coleta de todos os pesos na rede, b todos os bias (viés), n é o número total de entradas de treinamento, a é o vetor de saídas da rede (quando x é entrada) e a soma é sobre todas as entradas de treinamento x. Claro, a saída a depende de x, w e b, mas para manter a notação simples, eu não indiquei explicitamente essa dependência. A notação ||v|| apenas indica a função de comprimento usual para um vetor v. Chamaremos C a função de custo quadrático, que também é conhecido como o erro quadrático médio ou apenas o MSE (Mean Squared Error). Inspecionando a forma da função de custo quadrático, vemos que C (w, b) não é negativo, pois cada termo na soma não é negativo. Além disso, o custo C (w, b) torna-se pequeno, isto é, C (w, b) ≈ 0, precisamente quando y(x) é aproximadamente igual à saída, a, para todas as entradas de treinamento x.

Portanto, nosso algoritmo de treinamento faz um bom trabalho se ele pode encontrar pesos e bias para que C (w, b) ≈ 0. Isso significa basicamente que nosso modelo fez as previsões corretas, ou seja, cada vez que apresentamos ao modelo uma imagem com dígito 3, ele é capaz de reconhecer que se trata do número 3.

Em contraste, o algoritmo não terá boa performance, quando C (w, b) for um valor maior que 0 – isso significaria que nosso algoritmo não está conseguindo fazer as previsões, ou seja, quando apresentado a imagem com o dígito 3, ele não é capaz de prever que se trata de um número 3. Isso ocorre, porque a diferença entre o valor real da saída e o valor previsto pelo modelo, é muito alta. Assim, o objetivo do nosso algoritmo de treinamento será minimizar o custo C(w, b) em função dos pesos e dos bias. Em outras palavras, queremos encontrar um conjunto de pesos e bias que tornem o custo o menor possível. Vamos fazer isso usando um algoritmo conhecido como Descida do Gradiente (Gradient Descent).

Mas antes, uma pergunta. Por que introduzir o custo quadrático? Afinal, não nos interessamos principalmente pelo número de imagens corretamente classificadas pela rede? Por que não tentar maximizar esse número diretamente, em vez de minimizar uma medida, como o custo quadrático? O problema com isso é que o número de imagens corretamente classificadas não é uma “smooth function” dos pesos e bias na rede. Geralmente, fazer pequenas mudanças nos pesos e bias não causará nenhuma alteração no número de imagens de treinamento classificadas corretamente. Isso torna difícil descobrir como mudar os pesos e os bias para melhorar o desempenho. Se, em vez disso, usamos uma “smooth cost function”, como o custo quadrático, revela-se fácil descobrir como fazer pequenas mudanças nos pesos e nos bias para obter uma melhoria no custo. É por isso que nos concentramos primeiro na minimização do custo quadrático e somente depois examinaremos a precisão da classificação.

Mesmo considerando que queremos usar uma “smooth cost function”, você ainda pode se perguntar por que escolhemos a função quadrática. Talvez se escolhêssemos uma função de custo diferente, obteríamos um conjunto totalmente diferente de pesos e bias? Esta é uma preocupação válida e, mais tarde, revisitaremos a função de custo e faremos algumas modificações. No entanto, a função de custo quadrático mostrada anteriormente funciona perfeitamente para entender os conceitos básicos de aprendizagem em redes neurais, então ficaremos com isso por enquanto.

Recapitulando, nosso objetivo na construção de uma rede neural é encontrar pesos e bias que minimizem a função de custo quadrático C (w, b).

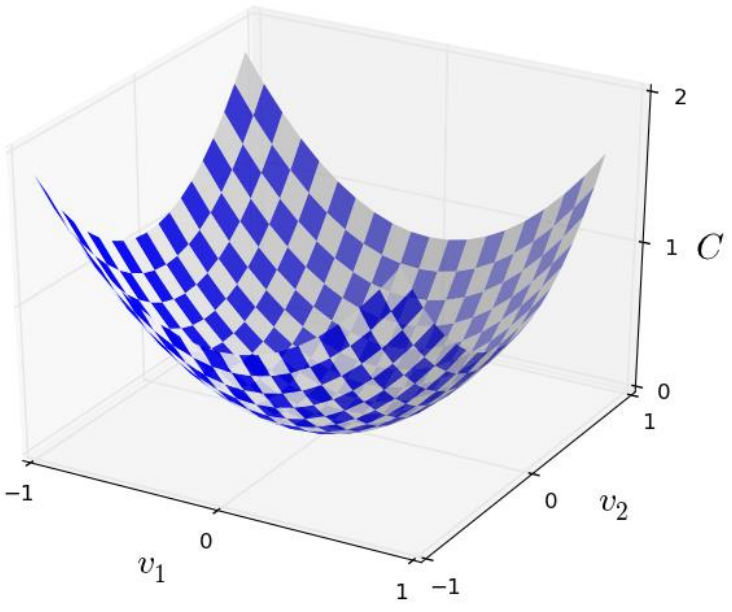
Descida do Gradiente

A maioria das tarefas em Machine Learning são na verdade problemas de otimização e um dos algoritmos mais usados para isso é o Algoritmo de Descida do Gradiente. Para um iniciante, o nome Algoritmo de Descida do Gradiente pode parecer intimidante, mas espero que depois de ler o que está logo abaixo, isso deixe de ser um mistério para você.

A Descida do Gradiente é uma ferramenta padrão para otimizar funções complexas iterativamente dentro de um programa de computador. Seu objetivo é: dada alguma função arbitrária, encontrar um mínimo. Para alguns pequenos subconjuntos de funções – aqueles que são convexos – há apenas um único *minumum* que também acontece de ser global. Para as funções mais realistas, pode haver muitos mínimos, então a maioria dos mínimos são locais. Certifique-se de que a otimização encontre o "melhor" *minimum* e não fique preso em mínimos sub-otimistas (um problema comum durante o treinamento do algoritmo).

Para compreender a intuição da Descida do Gradiente, vamos simplificar um pouco as coisas. Vamos imaginar que simplesmente recebemos uma função de muitas variáveis e queremos minimizar essa função. Vamos desenvolver a técnica chamada Descida do Gradiente que pode ser usada para resolver tais problemas de minimização. Então, voltaremos para a função específica que queremos minimizar para as redes neurais.

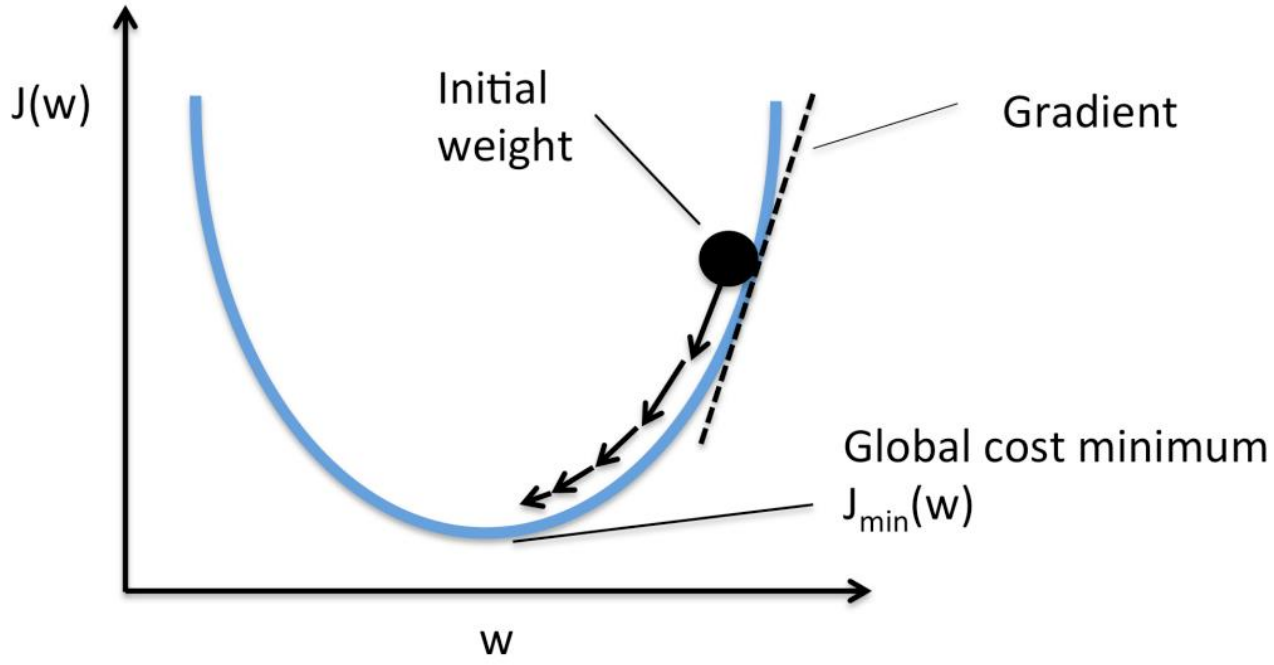
Ok, suponhamos que estamos tentando minimizar alguma função, C(v). Esta poderia ser qualquer função de valor real de muitas variáveis, onde v = v1, v2, ... Observe que eu substitui a notação w e b por v para enfatizar que esta poderia ser qualquer função – não estamos mais pensando especificamente no contexto das redes neurais apenas. Para minimizar C (v), vamos imaginar C como uma função de apenas duas variáveis, que chamaremos v1 e v2, conforme pode ser visto na figura abaixo:



O que queremos é encontrar onde C atinge seu mínimo global. Fica claro, que para a função traçada no gráfico acima, podemos observar facilmente o gráfico e encontrar o mínimo. Mas uma função geral, C, pode ser uma função complicada de muitas variáveis, e geralmente não será possível apenas observar o gráfico para encontrar o mínimo.

Uma maneira de atacar o problema é usar Cálculo (especificamente Álgebra Linear) para tentar encontrar o mínimo de forma analítica. Podemos calcular derivadas e depois tentar usá-las para encontrar lugares onde C é um *extremum*. Isso pode funcionar quando C é uma função de apenas uma ou algumas variáveis. Mas vai se transformar em um pesadelo quando tivermos muitas outras variáveis. E para as redes neurais, muitas vezes queremos muito mais variáveis – as maiores redes neurais têm funções de custo que dependem de bilhões de pesos e bias de uma maneira extremamente complicada. Usando "apenas" Cálculo para minimizar isso, não funcionará e precisamos de algo mais! Precisamos de um algoritmo de otimização capaz de minimizar C (v).

Felizmente, há uma analogia que nos ajuda a compreender como encontrar a solução. Começamos por pensar em nossa função como uma espécie de vale e imaginamos uma bola rolando pela encosta do vale, conforme pode ser visto na figura abaixo. Nossa experiência diária nos diz que a bola acabará rolando para o fundo do vale. Talvez possamos usar essa ideia como forma de encontrar um mínimo para a função? Escolheríamos aleatoriamente um ponto de partida para uma bola (imaginária), e então simularíamos o movimento da bola enquanto ela rola até o fundo do vale. Poderíamos fazer essa simulação simplesmente por derivadas de computação da função C – essas derivadas nos diriam tudo o que precisamos saber sobre a "forma" local do vale, e, portanto, como nossa bola deve rolar.



Representação da Descida do Gradiente (com o objetivo de minimizar a função de custo)

Ou seja, a Descida do Gradiente é um algoritmo de otimização usado para encontrar os valores

de parâmetros (coeficientes ou se preferir w e b – weight e bias) de uma função que minimizam uma função de custo. A Descida do Gradiente é melhor usada quando os parâmetros não podem ser calculados analiticamente (por exemplo, usando álgebra linear) e devem ser pesquisados por um algoritmo de otimização.

O procedimento começa com valores iniciais para o coeficiente ou coeficientes da função. Estes poderiam ser 0.0 ou um pequeno valor aleatório (a inicialização dos coeficiente é parte crítica do processo e diversas técnicas podem ser usadas, ficando a escolha a cargo do Cientista de Dados e do problema a ser resolvido com o modelo). Poderíamos iniciar assim nossos coeficientes (valores de w e b):

$$coeficiente = 0,0$$

O custo dos coeficientes é avaliado ligando-os à função e calculando o custo.

$$custo = f(coeficiente)$$

ou

$$custo = avaliar(f(coeficiente))$$

A derivada do custo é calculada. A derivada é um conceito de Cálculo e refere-se à inclinação da função em um determinado ponto. Precisamos conhecer a inclinação para que possamos conhecer a direção (sinal) para mover os valores dos coeficientes para obter um custo menor na próxima iteração.

$$delta = derivado(custo)$$

Agora que sabemos da derivada em que direção está em declive, podemos atualizar os valores dos coeficientes. Um parâmetro de taxa de aprendizagem (alfa) deve ser especificado e controla o quanto os coeficientes podem mudar em cada atualização.

$$coeficiente = coeficiente - (alfa * delta)$$

Este processo é repetido até que o **custo dos coeficientes** (**função de custo**) seja 0,0 ou próximo o suficiente de zero, indicando que as saídas da rede estão cada vez mais próximas dos valores reais (saídas desejadas).

A Descida do Gradiente é simples, mas exige que seja calculado o gradiente da função de custo ou a função que você está otimizando, mas além disso, é muito direto. Em resumo:

> Você divide seus dados em amostras e a cada amostra (sample), você passa as entradas pela rede, multiplica pelos pesos, soma, e no final você vai ter sua saida (a previsão da rede). Você então compara a saída da sua rede com o a resposta certa, calcula o erro, e então retroage esse erro (backpropagation), ajustando os pesos de cada neurônio de cada camada. Quando você acabar de fazer a atualização dos pesos, uma nova amostra é introduzida e ela será multiplicada pelos pesos já atualizados. Esse processo de atualizar os pesos é que é chamado de "aprendizado".
>
> Se você observar os algoritmos mais atuais, todos trabalham dentro de um conceito relativamente novo chamado de mini-lotes (mini-batches). Para otimizar a performance, o que se faz é passar pela rede múltiplas amostras (por exemplo 128 amostras), calcular o erro médio delas e então realizar o backpropagation e a atualização dos pesos. Do ponto de vista da atualização dos pesos, 1 amostra $=$ 128 amostras. Esse é um conceito mais novo, necessário principalmente no treinamento de grandes modelos de Deep Learning.

Em seguida, veremos como podemos usar isso em algoritmos de aprendizado de máquina.

Batch Gradient Descent em Aprendizado de Máquina

O objetivo de todos os algoritmos supervisionados de aprendizagem de máquina é estimar uma função de destino (f) que mapeia dados de entrada (X) para as variáveis de saída (Y). Isso descreve todos os problemas de classificação e regressão (aprendizagem supervisionada).

Alguns algoritmos de aprendizagem de máquina têm coeficientes que caracterizam a estimativa de algoritmos para a função alvo (f). Diferentes algoritmos têm diferentes representações e diferentes coeficientes, mas muitos deles requerem um processo de otimização para encontrar o conjunto de coeficientes que resultam na melhor estimativa da função alvo. Os exemplos comuns de algoritmos com coeficientes que podem ser otimizados usando descida do gradiente são Regressão linear e Regressão logística.

A avaliação de quão próximo um modelo de aprendizagem de máquina estima a função de destino pode ser calculada de várias maneiras, muitas vezes específicas para o algoritmo de aprendizagem de máquina. A função de custo envolve a avaliação dos coeficientes no modelo de aprendizagem de máquina calculando uma previsão para o modelo para cada instância de treinamento no conjunto de dados e comparando as previsões com os valores de saída reais e calculando uma soma ou erro médio (como a Soma de Residuais Quadrados ou SSR no caso de regressão linear).

A partir da função de custo, uma derivada pode ser calculada para cada coeficiente para que ele possa ser atualizado usando exatamente a equação de atualização descrita acima.

O custo é calculado para um algoritmo de aprendizado de máquina em todo o conjunto de dados de treinamento para cada iteração do algoritmo de descida de gradiente. Uma iteração do

algoritmo é chamada de um lote e esta forma de descida do gradiente é referida como descida do gradiente em lote (Batch Gradient Descent).

A descida do gradiente em lote é a forma mais comum de descida do gradiente em Machine Learning.

Stochastic Gradient Descent em Aprendizado de Máquina

A Descida do Gradiente pode ser lenta para executar em conjuntos de dados muito grandes. Como uma iteração do algoritmo de descida do gradiente requer uma previsão para cada instância no conjunto de dados de treinamento, pode demorar muito quando você tem muitos milhões de instâncias.

Em situações em que você possui grandes quantidades de dados, você pode usar uma variação da descida do gradiente chamada Stochastic Gradient Descent.

Nesta variação, o procedimento de descida do gradiente descrito acima é executado, mas a atualização para os coeficientes é realizada para cada instância de treinamento, em vez do final do lote de instâncias.

O primeiro passo do procedimento exige que a ordem do conjunto de dados de treinamento seja randomizada. Isto é, misturar a ordem que as atualizações são feitas para os coeficientes. Como os coeficientes são atualizados após cada instância de treinamento, as atualizações serão barulhentas saltando por todo o lado, e assim o custo correspondente funcionará. Ao misturar a ordem para as atualizações dos coeficientes, ela aproveita essa caminhada aleatória e evita que ela fique "distraída" ou presa.

O procedimento de atualização para os coeficientes é o mesmo que o anterior, exceto que o custo não é somado em todos os padrões de treinamento, mas sim calculado para um padrão de treinamento.

A aprendizagem pode ser muito mais rápida com descida de gradiente estocástica para conjuntos de dados de treinamento muito grandes e muitas vezes você só precisa de um pequeno número de passagens através do conjunto de dados para alcançar um conjunto de coeficientes bom o suficiente.

Ufa, você ainda está aí? Entende agora porque Cientistas de Dados e Engenheiros de IA devem ser muito bem remunerados? Eles são os "magos" que estão ajudando a transformar o mundo com Machine Learning. E este capítulo foi apenas uma breve introdução! Voltaremos a este assunto mais a frente no livro, quando estudarmos outros algoritmos. Mas caso você queira aprender em detalhes como tudo isso funciona e criar seus modelos usando linguagens R, Python, Scala ou Java, para aplicações comerciais, confira: Machine Learning, Machine Learning com Scala e Spark, Deep Learning e Análise Preditiva com Machine Learning em Java.

Tenho certeza que você está ansioso para criar e treinar sua primeira rede neural. Então, não perca o próximo capítulo!

Referências:

Formação Inteligência Artificial

Machine Learning

Machine Learning com Scala e Spark

Análise Preditiva com Machine Learning em Java

MNIST

Derivada

Machine Learning

The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition

Gradient Descent For Machine Learning

Pattern Recognition and Machine Learning

Understanding Activation Functions in Neural Networks

Redes Neurais, princípios e práticas

Neural Networks and Deep Learning

An overview of gradient descent optimization algorithms

Optimization: Stochastic Gradient Descent

Gradient Descent vs Stochastic Gradient Descent vs Mini-Batch Learning

CAPÍTULO 13

Construindo uma Rede Neural com Linguagem Python

Ok. Chegou a hora. Vamos escrever um programa em linguagem Python que aprenda como reconhecer dígitos manuscritos, usando Stochastic Gradient Descent e o dataset de treinamento MNIST. Se você chegou até aqui sem ler os capítulos anteriores, então pare imediatamente, leia os últimos 12 capítulos e depois volte aqui! Não tenha pressa! Não existe atalho para o aprendizado!

********** Atenção **********

Este capítulo considera que você já tem o interpretador Python (versão 3.6.x) instalado no seu computador, seja ele com sistema operacional Windows, MacOS ou Linux. Recomendamos que você instale o Anaconda e que já possua conhecimentos em linguagem Python. Se esse não for seu caso, antes de ler este capítulo e executar os exemplos aqui fornecidos, acesse o curso gratuito Python Fundamentos Para Análise de Dados.

Usaremos Python 3 e os scripts podem ser encontrados no repositório do livro no GitHub. Vamos começar!

****

Quando descrevemos o dataset MNIST anteriormente, dissemos que ele estava dividido em 60.000 imagens de treinamento e 10.000 imagens de teste. Essa é a descrição oficial do MNIST. Mas vamos dividir os dados de forma um pouco diferente. Deixaremos as imagens de teste como está, mas dividiremos o conjunto de treinamento MNIST de 60.000 imagens em duas partes: um conjunto de 50.000 imagens, que usaremos para treinar nossa rede neural e um conjunto separado de validação de 10.000 imagens. Não utilizaremos os dados de validação neste capítulo, porém mais tarde, aqui mesmo no livro, usaremos este dataset quando estivermos configurando certos hiperparâmetros da rede neural, como a taxa de aprendizado por exemplo. Embora os dados de validação não façam parte da especificação MNIST original, muitas pessoas usam o MNIST desta forma e o uso de dados de validação é comum em redes neurais. Quando eu me referir aos "dados de treinamento MNIST" de agora em diante, vou me referir ao nosso conjunto de dados de 50.000 imagens, e não ao conjunto de dados de 60.000 imagens. Fique atento!

Além dos dados MNIST, também precisamos de uma biblioteca Python chamada Numpy, para álgebra linear. Se você instalou o Anaconda, não precisa se preocupar, pois o Numpy já está instalado. Caso contrário, será necessário fazer a instalação do pacote.

Mas antes de carregar e dividir os dados, vamos compreender os principais recursos do nosso código para construção de uma rede neural. A peça central é uma classe chamada **Network**, que usamos para representar uma rede neural.

Abaixo a classe Network e seu construtor:

```
class Network(object):

    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x) for x, y in zip(sizes[:-1], sizes[1:])]
```

Neste código, o parâmetro **sizes** contêm o número de neurônios nas respectivas camadas, sendo um objeto do tipo lista em Python. Então, por exemplo, se queremos criar um objeto da classe Network com 2 neurônios na primeira camada, 3 neurônios na segunda camada e 1 neurônio na camada final, aqui está o código que usamos para instanciar um objeto da classe Network:

rede1 = Network([2, 3, 1])

Os bias e pesos no objeto rede1 são todos inicializados aleatoriamente, usando a função Numpy **np.random.randn** para gerar distribuições gaussianas com 0 de média e desvio padrão 1. Esta inicialização aleatória dá ao nosso algoritmo de descida do gradiente estocástico um local para começar. Em capítulos posteriores, encontraremos melhores maneiras de inicializar os pesos e os bias. Observe que o código de inicialização de rede assume que a primeira camada de neurônios é uma camada de entrada e omite a definição de quaisquer bias para esses neurônios, uma vez que os bias são usados apenas para calcular as saídas de camadas posteriores.

Observe também que os bias e pesos são armazenados como listas de matrizes Numpy. Assim, por exemplo, rede1.weights[1] é uma matriz Numpy armazenando os pesos conectando a segunda e terceira camadas de neurônios. (Não é a primeira e segunda camadas, uma vez que a indexação da lista em Python começa em 0.) Uma vez que rede1.weights[1] é bastante detalhado, vamos apenas indicar essa matriz w. É uma matriz tal que wjk é o peso para a conexão entre o neurônio kth na segunda camada e o neurônio jth na terceira camada. Essa ordenação dos índices j e k pode parecer estranha – certamente teria mais sentido trocar os índices j e k? A grande vantagem de usar essa ordenação é que isso significa que o vetor de ativações da terceira camada de neurônios é:

$$a' = \sigma(wa + b)$$

Equação 1

Onde, **a** é o vetor de ativações da segunda camada de neurônios.

Para obter um **a'** multiplicamos **a** pela matriz de peso **w**, e adicionamos o vetor **b** com os bias (se você leu os capítulos anteriores, isso não deve ser novidade agora). Em seguida, aplicamos a função **σ** de forma elementar a cada entrada no vetor **wa + b**. (Isto é chamado de vetorizar a função σ.)

Com tudo isso em mente, é fácil escrever código que computa a saída de uma instância de rede. Começamos definindo a função sigmoide:

```
# Função de Ativação Sigmóide
def sigmoid(z):
    return 1.0/(1.0+np.exp(-z))
```

Observe que quando a entrada z é um vetor ou uma matriz Numpy, Numpy aplica automaticamente a função sigmoid elementwise, ou seja, na forma vetorizada.

Em seguida, adicionamos um método feedforward à classe Network, que, dada a entrada a para a rede, retorna a saída correspondente. Basicamente o método feedforward aplica a Equação 1 mostrada acima, para cada camada:

```
class Network(object):

    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x) for x, y in zip(sizes[:-1], sizes[1:])]

    def feedforward(self, a):
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a
```

A principal atividade que queremos que nossos objetos da classe Network façam é aprender. Para esse fim, criaremos um método SGD (Stochastic Gradient Descent). Aqui está o código. É um pouco misterioso em alguns lugares, mas vamos explicar em detalhes mais abaixo:

```
class Network(object):

    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x) for x, y in zip(sizes[:-1], sizes[1:])]

    def feedforward(self, a):
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a

    def SGD(self, training_data, epochs, mini_batch_size, eta, test_data=None):
        training_data = list(training_data)
        n = len(training_data)

        if test_data:
            test_data = list(test_data)
            n_test = len(test_data)

        for j in range(epochs):
            random.shuffle(training_data)
            mini_batches = [training_data[k:k+mini_batch_size] for k in range(0, n, mini_batch_size)]

            for mini_batch in mini_batches:
                self.update_mini_batch(mini_batch, eta)

            if test_data:
                print("Epoch {} : {} / {}".format(j,self.evaluate(test_data),n_test));
            else:
                print("Epoch {} finalizada".format(j))
```

O **training_data** é uma lista de tuplas (x, y) que representam as entradas de treinamento e as correspondentes saídas desejadas. As variáveis **epochs** e **mini_batch_size** são o que você esperaria – o número de épocas para treinar e o tamanho dos mini-lotes a serem usados durante a amostragem, enquanto **eta** é a taxa de aprendizagem, η. Se o argumento opcional test_data for fornecido, o programa avaliará a rede após cada período de treinamento e imprimirá progresso parcial. Isso é útil para rastrear o progresso, mas retarda substancialmente as coisas.

O código funciona da seguinte forma. Em cada época, ele começa arrastando aleatoriamente os dados de treinamento e, em seguida, particiona-os em mini-lotes de tamanho apropriado. Esta é uma maneira fácil de amostragem aleatória dos dados de treinamento. Então, para cada mini_batch, aplicamos um único passo de descida do gradiente. Isso é feito pelo código self.update_mini_batch (mini_batch, eta), que atualiza os pesos e os bias da rede de acordo com uma única iteração de descida de gradiente, usando apenas os dados de treinamento em mini_batch. Aqui está o código para o método update_mini_batch:

```
class Network(object):

    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x) for x, y in zip(sizes[:-1], sizes[1:])]

    def feedforward(self, a):
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a

    def SGD(self, training_data, epochs, mini_batch_size, eta, test_data=None):
        training_data = list(training_data)
        n = len(training_data)

        if test_data:
            test_data = list(test_data)
            n_test = len(test_data)

        for j in range(epochs):
            random.shuffle(training_data)
            mini_batches = [
                training_data[k:k+mini_batch_size]
                for k in range(0, n, mini_batch_size)]
            for mini_batch in mini_batches:
                self.update_mini_batch(mini_batch, eta)
            if test_data:
                print("Epoch {} : {} / {}".format(j,self.evaluate(test_data),n_test));
            else:
                print("Epoch {} finalizada".format(j))

    def update_mini_batch(self, mini_batch, eta):

        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]

        for x, y in mini_batch:
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)
            nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
            nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]

        self.weights = [w-(eta/len(mini_batch))*nw for w, nw in zip(self.weights, nabla_w)]
        self.biases = [b-(eta/len(mini_batch))*nb for b, nb in zip(self.biases, nabla_b)]
```

A maior parte do trabalho é feita pela linha delta_nabla_b, delta_nabla_w = self.backprop (x, y). Isso invoca algo chamado algoritmo de backpropagation, que é uma maneira rápida de calcular o gradiente da função de custo. Portanto, update_mini_batch funciona simplesmente calculando esses gradientes para cada exemplo de treinamento no mini_batch e, em seguida, atualizando self.weights e self.biases adequadamente.

Abaixo você encontra o código para self.backprop, mas não estudaremos ele agora. Estudaremos em detalhes como funciona o backpropagation no próximo capítulo, incluindo o código para self.backprop. Por hora, basta assumir que ele se comporta conforme indicado, retornando o gradiente apropriado para o custo associado ao exemplo de treinamento x.

```
def backprop(self, x, y):

    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]

    # Feedforward
    activation = x

    # Lista para armazenar todas as ativações, camada por camada
    activations = [x]

    # Lista para armazenar todos os vetores z, camada por camada
    zs = []

    for b, w in zip(self.biases, self.weights):
        z = np.dot(w, activation)+b
        zs.append(z)
        activation = sigmoid(z)
        activations.append(activation)

    # Backward pass
    delta = self.cost_derivative(activations[-1], y) * sigmoid_prime(zs[-1])
    nabla_b[-1] = delta
    nabla_w[-1] = np.dot(delta, activations[-2].transpose())

    for l in range(2, self.num_layers):
        z = zs[-l]
        sp = sigmoid_prime(z)
        delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
        nabla_b[-l] = delta
        nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
    return (nabla_b, nabla_w)
```

No programa completo disponível no Github você encontra comentários explicando como ocorre todo o processo. Além do self.backprop, o programa é auto-explicativo – todo o levantamento pesado é feito em self.SGD e self.update_mini_batch, que já discutimos. O método self.backprop faz uso de algumas funções extras para ajudar no cálculo do gradiente, nomeadamente sigmoid_prime, que calcula a derivada da função σ e self.cost_derivative.

A classe Network é em essência nosso algoritmo de rede neural. A partir dela criamos uma instância (como rede1), alimentamos com os dados de treinamento e realizamos o treinamento. Avaliamos então a performance da rede com dados de teste e repetimos todo o processo até alcançar o nível de acurácia desejado em nosso projeto. Quando o modelo final estiver pronto, usamos para realizar as previsões para as quais o modelo foi criado, apresentando a ele novos conjuntos de dados e extraindo as previsões. Perceba que este é um algoritmo de rede neural bem simples, mas que permite compreender como funcionam as redes neurais e mais tarde, aqui mesmo no livro, as redes neurais profundas ou Deep Learning.

No próximo capítulo vamos continuar trabalhando com este algoritmo e compreender como funciona o Backpropagation. Na sequência, vamos carregar os dados, treinar e testar nossa rede neural e então usá-la para reconhecer dígitos manuscritos. Até lá.

Referências:

Formação Inteligência Artificial

Python Fundamentos Para Análise de Dados

Machine Learning

Machine Learning com Scala e Spark

Análise Preditiva com Machine Learning em Java

MNIST

Derivada

Machine Learning

The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition

Gradient Descent For Machine Learning

Pattern Recognition and Machine Learning

Understanding Activation Functions in Neural Networks

Redes Neurais, princípios e práticas

Neural Networks and Deep Learning

An overview of gradient descent optimization algorithms

Optimization: Stochastic Gradient Descent

Gradient Descent vs Stochastic Gradient Descent vs Mini-Batch Learning

CAPÍTULO 14

Algoritmo Backpropagation Parte 1 – Grafos Computacionais e Chain Rule

No último capítulo, vimos como as redes neurais podem aprender seus pesos e bias usando o algoritmo de gradiente descendente. Houve, no entanto, uma lacuna na nossa explicação: não discutimos como calcular o gradiente da função de custo. Neste capítulo, explicaremos sobre um algoritmo usado para calcular esses gradientes, um algoritmo conhecido como backpropagation. Como esse tema é a essência do treinamento de redes neurais, vamos dividí-lo em dois capítulos. Vamos começar com Algoritmo Backpropagation Parte 1 – Grafos Computacionais e Chain Rule.

O backpropagation é indiscutivelmente o algoritmo mais importante na história das redes neurais – sem backpropagation, seria quase impossível treinar redes de aprendizagem profunda da forma que vemos hoje. O backpropagation pode ser considerado a pedra angular das redes neurais modernas e consequentemente do Deep Learning.

O algoritmo backpropagation foi originalmente introduzido na década de 1970, mas sua importância não foi totalmente apreciada até um famoso artigo de 1986 de David Rumelhart, Geoffrey Hinton e Ronald Williams. Esse artigo descreve várias redes neurais em que o backpropagation funciona muito mais rapidamente do que as abordagens anteriores de aprendizado, possibilitando o uso de redes neurais para resolver problemas que antes eram insolúveis.

O backpropagation é o algoritmo-chave que faz o treinamento de modelos profundos algo computacionalmente tratável. Para as redes neurais modernas, ele pode tornar o treinamento com gradiente descendente até dez milhões de vezes mais rápido, em relação a uma implementação ingênua. Essa é a diferença entre um modelo que leva algumas horas ou dias para treinar e e outro que poderia levar anos (sem exagero).

Além de seu uso em Deep Learning, o backpropagation é uma poderosa ferramenta computacional em muitas outras áreas, desde previsão do tempo até a análise da estabilidade numérica. De fato, o algoritmo foi reinventado pelo menos dezenas de vezes em diferentes campos. O nome geral, independente da aplicação, é “diferenciação no modo reverso”.

Fundamentalmente, backpropagation é uma técnica para calcular derivadas rapidamente (não sabe o que é derivada? Consulte o link para um excelente vídeo em português explicando esse conceito em detalhes nas referências ao final deste capítulo). E é um truque essencial, não apenas em Deep Learning, mas em uma ampla variedade de situações de computação numérica. E para compreender backpropagation de forma efetiva, vamos primeiro compreender o conceito de grafo computacional e chain rule.

Grafo Computacional

Grafos computacionais são uma boa maneira de pensar em expressões matemáticas. O conceito de grafo foi introduzido por Leonhard Euler em 1736 para tentar resolver o problema das Pontes de Konigsberg. Grafos são modelos matemáticos para resolver problemas práticos do dia a dia, com várias aplicações no mundo real tais como: circuitos elétricos, redes de distribuição, relações de parentesco entre pessoas, análise de redes sociais, logística, redes de

estradas, redes de computadores e muito mais. Grafos são muito usados para modelar problemas em computação.

Um Grafo é um modelo matemático que representa relações entre objetos. Um grafo G = (V, E) consiste de um conjunto de vértices **V** (também chamados de nós), ligados por um conjunto de bordas ou arestas **E**. Para aprender sobre grafos em mais detalhes, clique aqui.

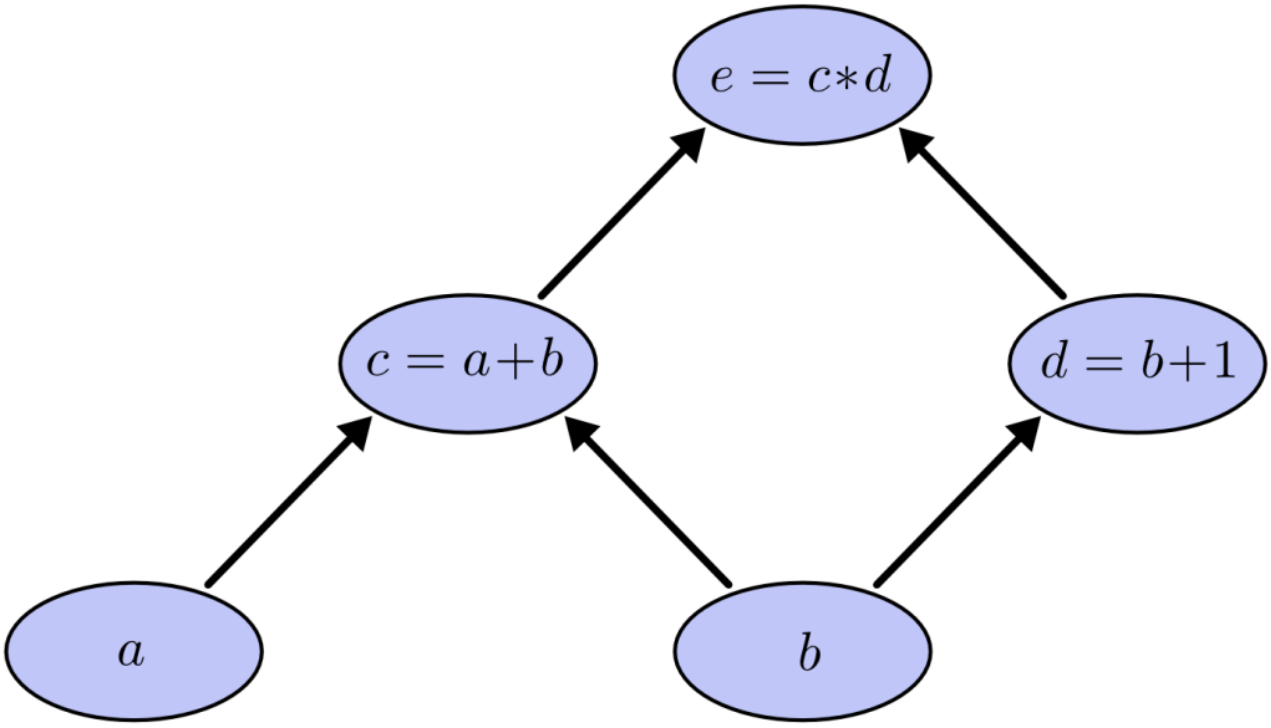
Por exemplo, considere a expressão:

$$e = (a + b) * (b + 1)$$

Existem três operações: duas adições e uma multiplicação. Para facilitar a compreensão sobre isso, vamos introduzir duas variáveis intermediárias c e d para que a saída de cada função tenha uma variável. Nós agora temos:

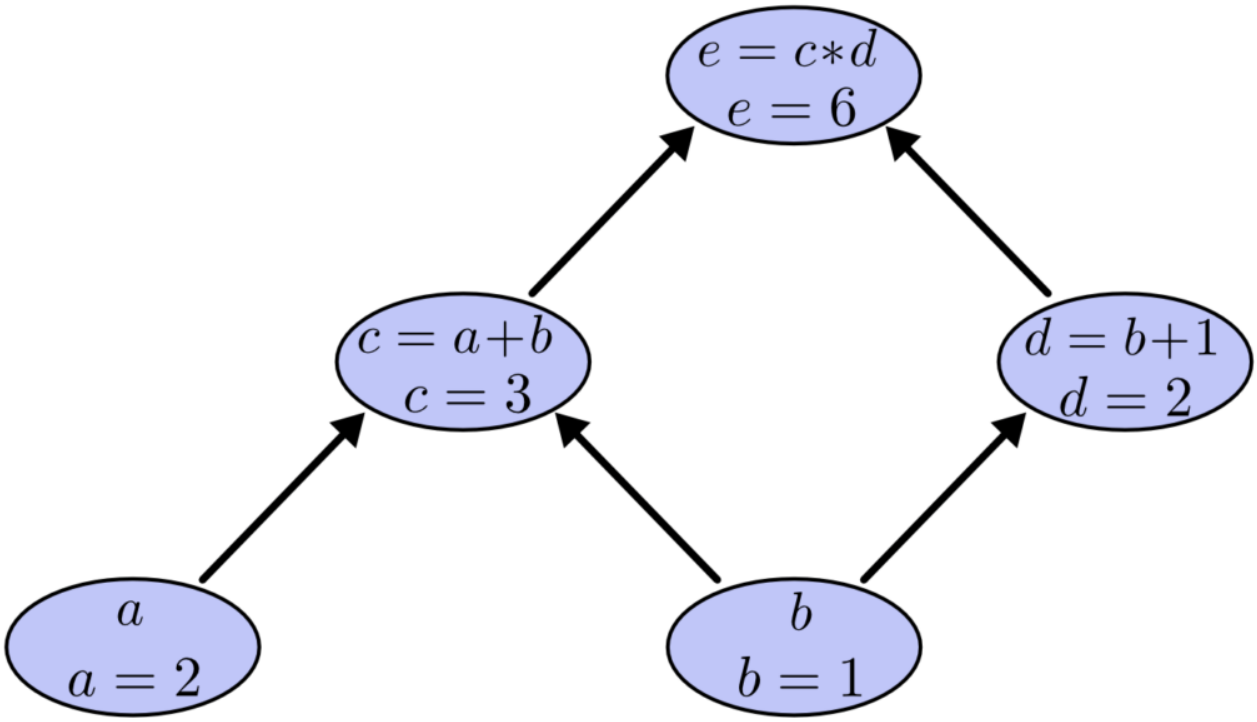
$$c = a+b$$
$$d = b+1$$
$$e = c*d$$

Para criar um grafo computacional, fazemos cada uma dessas operações nos ***nós***, juntamente com as variáveis de entrada. Quando o valor de um nó é a entrada para outro nó, uma seta vai de um para outro e temos nesse caso um grafo direcionado.



Esses tipos de grafos surgem o tempo todo em Ciência da Computação, especialmente ao falar sobre programas funcionais. Eles estão intimamente relacionados com as noções de grafos de dependência e grafos de chamadas. Eles também são a principal abstração por trás do popular framework de Deep Learning, o TensorFlow.

Podemos avaliar a expressão definindo as variáveis de entrada para determinados valores e computando os nós através do grafo. Por exemplo, vamos definir a = 2 e b = 1:



A expressão, nesse exemplo, é avaliada como 6.

Derivadas em Grafos Computacionais

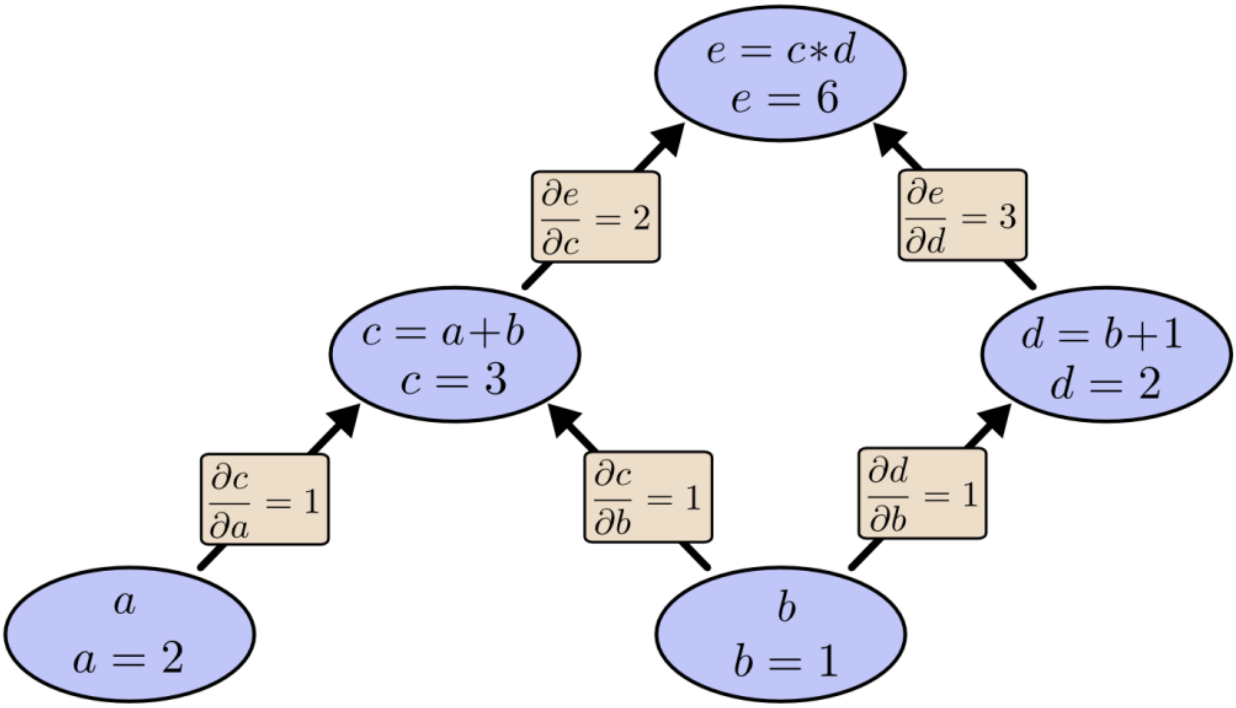
Se alguém quiser entender derivadas em um grafo computacional, a chave é entender as derivadas nas bordas (arestas que conectam os nós no grafo). Se **a** afeta diretamente **c**, então queremos saber como isso afeta **c**. Se **a** muda um pouco, como **c** muda? Chamamos isso de derivada parcial de **c** em relação a **a**.

Para avaliar as derivadas parciais neste grafo, precisamos da regra da soma e da regra do produto:

$$\frac{\partial}{\partial a}(a+b) = \frac{\partial a}{\partial a} + \frac{\partial b}{\partial a} = 1$$

$$\frac{\partial}{\partial u}uv = u\frac{\partial v}{\partial u} + v\frac{\partial u}{\partial u} = v$$

Abaixo, o grafo tem a derivada em cada borda (aresta) rotulada.



E se quisermos entender como os nós que não estão diretamente conectados afetam uns aos outros? Vamos considerar como **e** é afetado por **a**. Se mudarmos **a** uma velocidade de 1, **c** também muda a uma velocidade de 1. Por sua vez, **c** mudando a uma velocidade de 1 faz com que **e** mude a uma velocidade de 2. Então **e** muda a uma taxa de $1 * 2$ em relação a **a** (analise o diagrama acima para visualizar isso).

A regra geral é somar todos os caminhos possíveis de um nó para o outro, multiplicando as derivadas em cada aresta do caminho. Por exemplo, para obter a derivada de **e** em relação a **b**, obtemos:

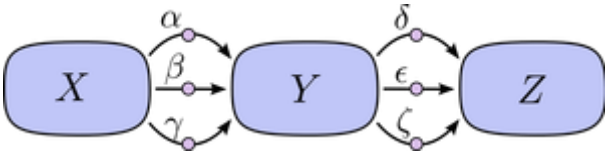
$$\frac{\partial e}{\partial b} = 1 * 2 + 1 * 3$$

Isso explica como **b** afeta **e** através de **c** e também como isso afeta **d**.

Essa regra geral de "soma sobre caminhos" é apenas uma maneira diferente de pensar sobre a regra da cadeia multivariada ou **chain rule**.

Fatorando os Caminhos

O problema com apenas "somar os caminhos" é que é muito fácil obter uma explosão combinatória no número de caminhos possíveis.



No diagrama acima, existem três caminhos de X a Y, e mais três caminhos de Y a Z. Se quisermos obter a derivada $\partial Z/\partial X$ somando todos os caminhos, precisamos calcular $3 * 3 = 9$ caminhos:

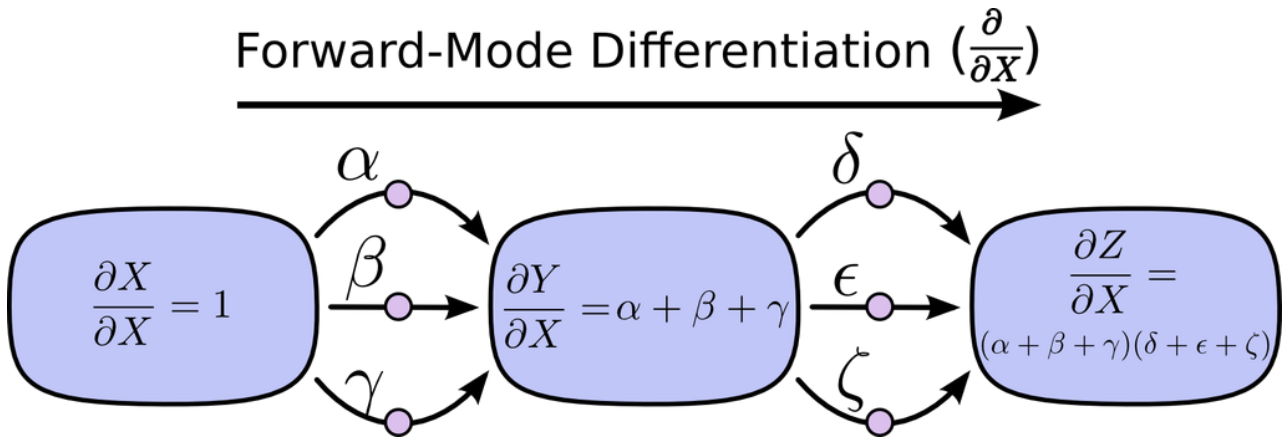
$$\frac{\partial Z}{\partial X} = \alpha\delta + \alpha\epsilon + \alpha\zeta + \beta\delta + \beta\epsilon + \beta\zeta + \gamma\delta + \gamma\epsilon + \gamma\zeta$$

O exemplo acima só tem nove caminhos, mas seria fácil o número de caminhos crescer exponencialmente à medida que o grafo se torna mais complicado. Em vez de apenas ingenuamente somar os caminhos, seria muito melhor fatorá-los:

$$\frac{\partial Z}{\partial X} = (\alpha + \beta + \gamma)(\delta + \epsilon + \zeta)$$

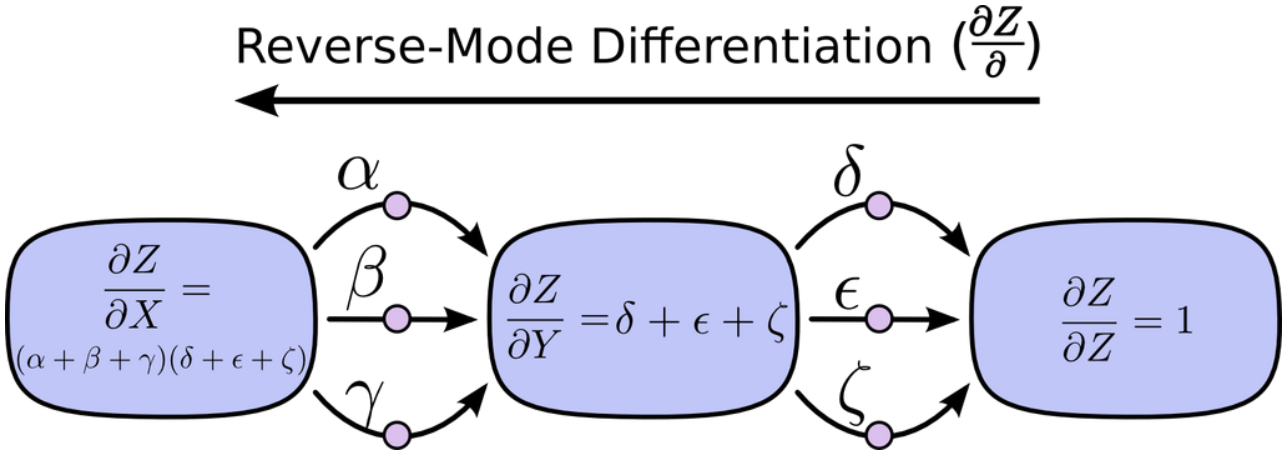
É aí que entram a "diferenciação de modo de avanço" (forward-mode differentiation ou forward pass) e a "diferenciação de modo reverso" (reverse-mode differentiation ou backpropagation). Eles são algoritmos para calcular a soma de forma eficiente fatorando os caminhos. Em vez de somar todos os caminhos explicitamente, eles calculam a mesma soma de forma mais eficiente, mesclando os caminhos juntos novamente em cada nó. De fato, os dois algoritmos tocam cada borda exatamente uma vez!

A diferenciação do modo de avanço inicia em uma entrada para o grafo e se move em direção ao final. Em cada nó, soma todos os caminhos que se alimentam. Cada um desses caminhos representa uma maneira na qual a entrada afeta esse nó. Ao adicioná-los, obtemos a maneira total em que o nó é afetado pela entrada, isso é a derivada.



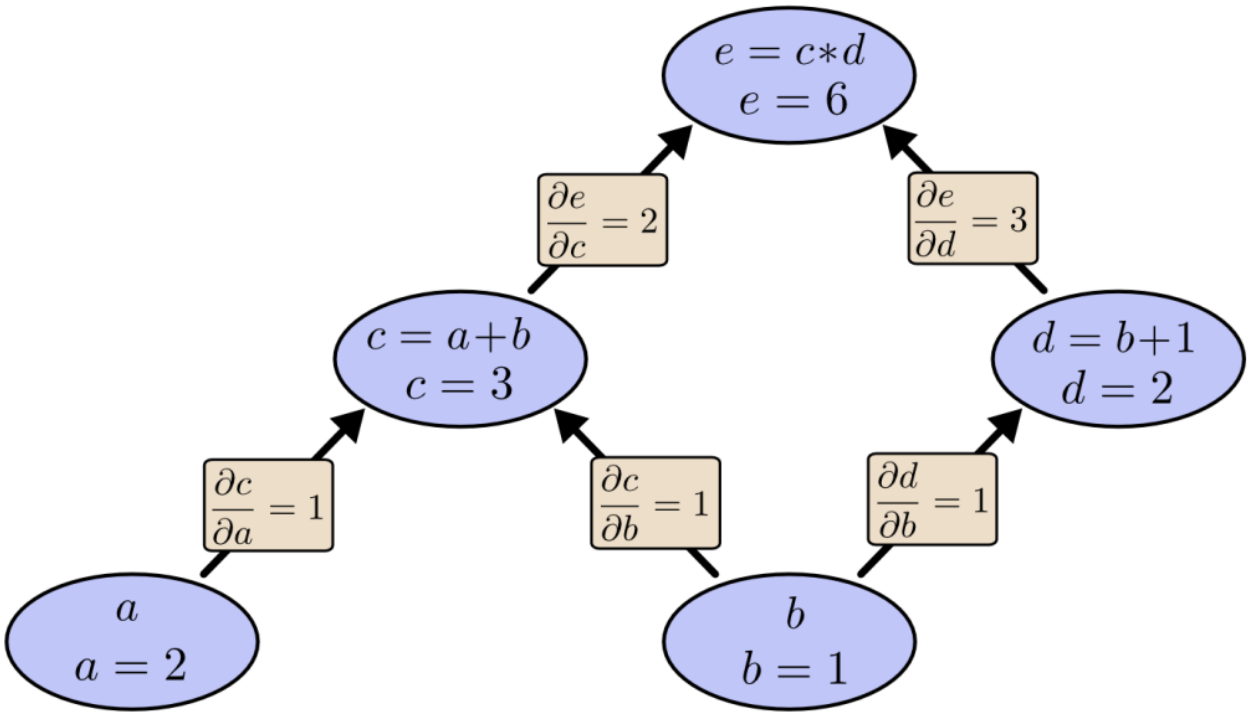
Embora você provavelmente não tenha pensado nisso em termos de grafos, a diferenciação no modo de avanço é muito parecida com o que você aprendeu implicitamente caso tenha feito alguma introdução a Cálculo.

A diferenciação no modo reverso, por outro lado, começa na saída do grafo e se move em direção ao início (ou seja, se retropropaga ou backpropagation). Em cada nó, ele mescla todos os caminhos originados nesse nó.

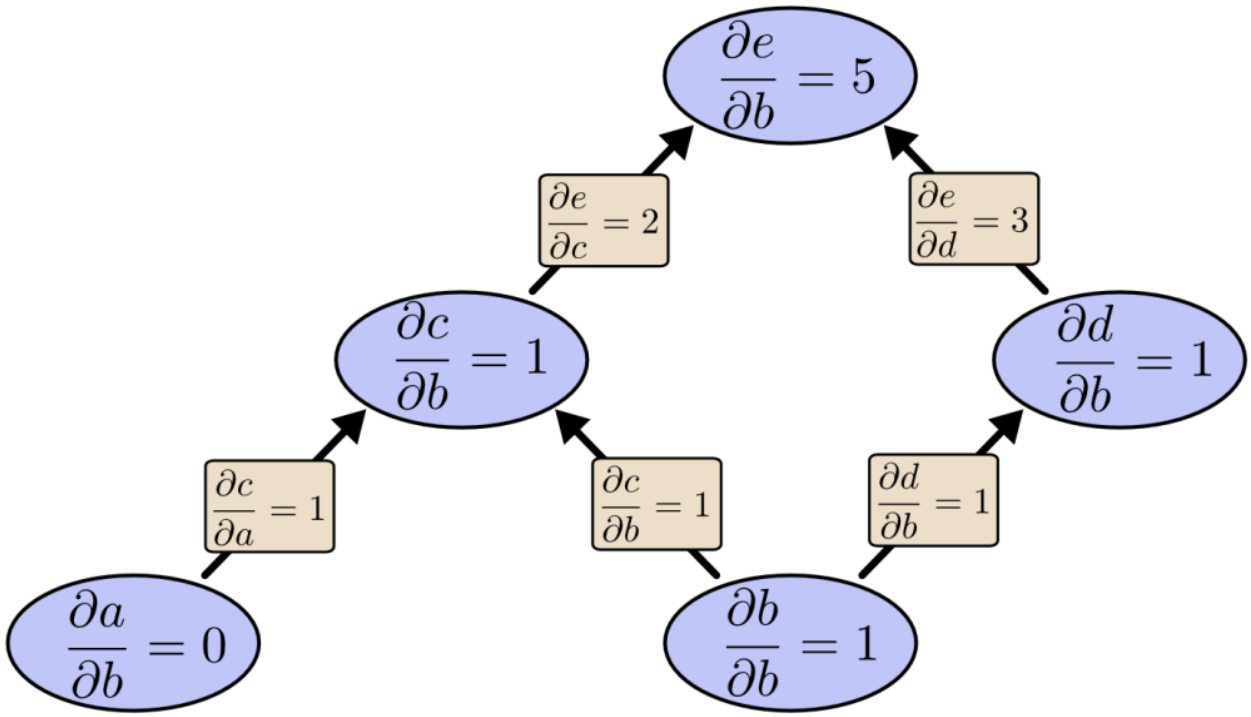


A diferenciação do modo de avanço rastreia como uma entrada afeta todos os nós. A diferenciação no modo reverso rastreia como cada nó afeta uma saída. Ou seja, a diferenciação de modo de avanço aplica o operador ∂/∂X a cada nó, enquanto a diferenciação de modo reverso aplica o operador ∂Z/∂ a cada nó. Se isso parece o conceito de programação dinâmica, é porque é exatamente isso! (acesse um material sobre programação dinâmica nas referências ao final do capítulo)

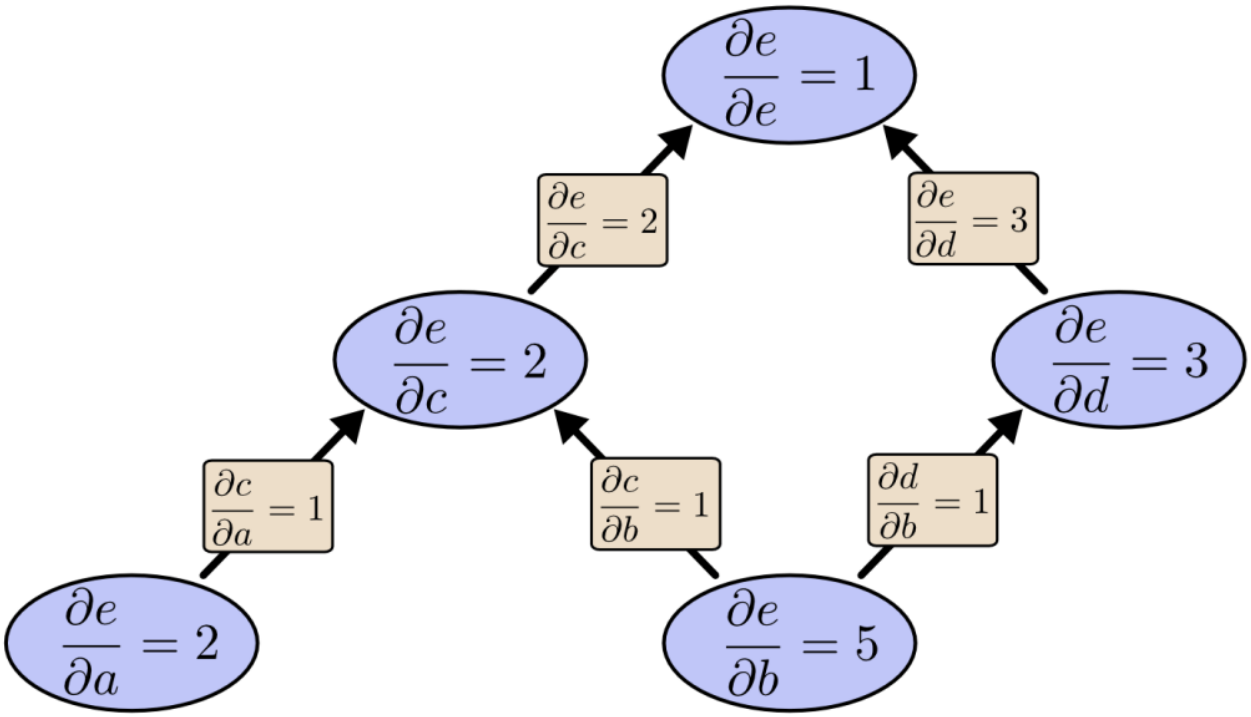
Nesse ponto, você pode se perguntar porque alguém se importaria com a diferenciação no modo reverso. Parece uma maneira estranha de fazer a mesma coisa que o modo de avanço. Existe alguma vantagem? Vamos considerar nosso exemplo original novamente:



Podemos usar a diferenciação de modo de avanço de **b** para cima. Isso nos dá a derivada de cada nó em relação a **b**.


$\frac{\partial e}{\partial b} = 5$
$\frac{\partial e}{\partial c} = 2$
$\frac{\partial e}{\partial d} = 3$
$\frac{\partial c}{\partial b} = 1$
$\frac{\partial d}{\partial b} = 1$
$\frac{\partial c}{\partial a} = 1$
$\frac{\partial c}{\partial b} = 1$
$\frac{\partial d}{\partial b} = 1$
$\frac{\partial a}{\partial b} = 0$
$\frac{\partial b}{\partial b} = 1$

Nós calculamos ∂e/∂b, a derivada de nossa saída em relação a um de nossos inputs. E se fizermos a diferenciação de modo reverso de **e** para baixo? Isso nos dá a derivada de **e** em relação a todos os nós:


$\frac{\partial e}{\partial e} = 1$
$\frac{\partial e}{\partial c} = 2$
$\frac{\partial e}{\partial d} = 3$
$\frac{\partial e}{\partial c} = 2$
$\frac{\partial e}{\partial d} = 3$
$\frac{\partial c}{\partial a} = 1$
$\frac{\partial c}{\partial b} = 1$
$\frac{\partial d}{\partial b} = 1$
$\frac{\partial e}{\partial a} = 2$
$\frac{\partial e}{\partial b} = 5$

Quando digo que a diferenciação no modo reverso nos dá a derivada de **e** em relação a cada nó, eu realmente quero dizer cada nó. Temos tanto ∂e/∂a quanto ∂e/∂b, as derivadas de **e** em relação a ambas as entradas. A diferenciação no modo de avanço nos deu a derivada de nossa saída em relação a uma única entrada, mas a diferenciação no modo reverso nos dá todos eles.

Para este grafo, isso é apenas um fator de duas velocidades, mas imagine uma função com um milhão de entradas e uma saída. A diferenciação no modo de avanço exigiria que passássemos pelo grafo um milhão de vezes para obter as derivadas. Diferenciação no modo reverso pode fazer isso em uma só passada! Uma aceleração de um fator de um milhão é bem legal e explica porque conseguimos treinar um modelo de rede neural profunda em tempo razoável.

Ao treinar redes neurais, pensamos no custo (um valor que descreve o quanto uma rede neural é ruim) em função dos parâmetros (números que descrevem como a rede se comporta). Queremos calcular as derivadas do custo em relação a todos os parâmetros, para uso em descida do gradiente. Entretanto, muitas vezes, há milhões ou até dezenas de milhões de parâmetros em uma rede neural. Então, a diferenciação no modo reverso, chamada de backpropagation no contexto das redes neurais, nos dá uma velocidade enorme!

Existem casos em que a diferenciação de modo de avanço faz mais sentido? Sim, existem! Onde o modo reverso fornece as derivadas de uma saída em relação a todas as entradas, o modo de avanço nos dá as derivadas de todas as saídas em relação a uma entrada. Se tiver uma função com muitas saídas, a diferenciação no modo de avanço pode ser muito, muito mais rápida.

Agora faz sentido?

Quando aprendemos pela primeira vez o que é backpropagation, a reação é: “Oh, essa é apenas a regra da cadeia (chain rule)! Como demoramos tanto tempo para descobrir?”

Na época em que o backpropagation foi inventado, as pessoas não estavam muito focadas nas redes neurais feedforward. Também não era óbvio que as derivadas eram o caminho certo para treiná-las. Esses são apenas óbvios quando você percebe que pode calcular rapidamente derivadas. Houve uma dependência circular.

Treinar redes neurais com derivadas? Certamente você ficaria preso em mínimos locais. E obviamente seria caro computar todas essas derivadas. O fato é que só porque sabemos que essa abordagem funciona é que não começamos imediatamente a listar os motivos que provavelmente não funcionaria. Já sabemos que funciona, mas novas abordagens vem sendo propostas no avanço das pesquisas em Deep Learning e Inteligência Artificial.

Conclusão da Parte 1

O backpropagation também é útil para entender como as derivadas fluem através de um modelo. Isso pode ser extremamente útil no raciocínio sobre porque alguns modelos são difíceis de otimizar. O exemplo clássico disso é o problema do desaparecimento de gradientes em redes neurais recorrentes, que discutiremos mais diante neste livro.

Por fim, há uma lição algorítmica ampla a ser retirada dessas técnicas. Backpropagation e forward-mode differentiation usam um poderoso par de truques (linearização e programação dinâmica) para computar derivadas de forma mais eficiente do que se poderia imaginar. Se você realmente entende essas técnicas, pode usá-las para calcular com eficiência várias outras expressões interessantes envolvendo derivadas.

Mas este capítulo teve como objetivo apenas ajudá-lo a compreender o algoritmo, já que praticamente não existe documentação sobre isso em português. Falta ainda compreender como o backpropagation é aplicado no treinamento das redes neurais. Ansioso por isso? Então acompanhe o próximo capítulo!

A Data Science Academy oferece um programa completo, onde esses e vários outros conceitos são estudados em detalhes e com várias aplicações práticas e usando TensorFlow. A Formação Inteligência Artificial é composta de 9 cursos, tudo 100% online e 100% em português, que aliam teoria e prática na medida certa, com aplicações reais de Inteligência Artificial. Confira o programa completo dos cursos: Formação Inteligência Artificial.

Referências:

Formação Inteligência Artificial

Me Salva! Cálculo – O que é uma derivada?

The Birth Of Graph Theory: Leonhard Euler And The Königsberg Bridge Problem

Learning representations by back-propagating errors

Chain Rule

Calculus on Computational Graphs: Backpropagation

How the backpropagation algorithm works

Dynamic programming

Nota: parte das imagens usadas neste capítulo foram extraídas no excelente post (citado nas referências acima) de Christopher Olah, pesquisador de Machine Learning do Google Brain, e com a devida autorização do autor.

CAPÍTULO 15

Algoritmo Backpropagation Parte 2 – Treinamento de Redes Neurais

O backpropagation é indiscutivelmente o algoritmo mais importante na história das redes neurais – sem backpropagation (eficiente), seria impossível treinar redes de aprendizagem profunda da forma que vemos hoje. O backpropagation pode ser considerado a pedra angular das redes neurais modernas e aprendizagem profunda. Neste capítulo, vamos compreender como o backpropagation é usado no treinamento das redes neurais: Algoritmo Backpropagation Parte 2 – Treinamento de Redes Neurais.

O algoritmo de backpropagation consiste em duas fases:

1. O passo para frente (forward pass), onde nossas entradas são passadas através da rede e as previsões de saída obtidas (essa etapa também é conhecida como fase de propagação).

2. O passo para trás (backward pass), onde calculamos o gradiente da função de perda na camada final (ou seja, camada de previsão) da rede e usamos esse gradiente para aplicar recursivamente a regra da cadeia (chain rule) para atualizar os pesos em nossa rede (etapa também conhecida como fase de atualização de pesos ou retro-propagação).

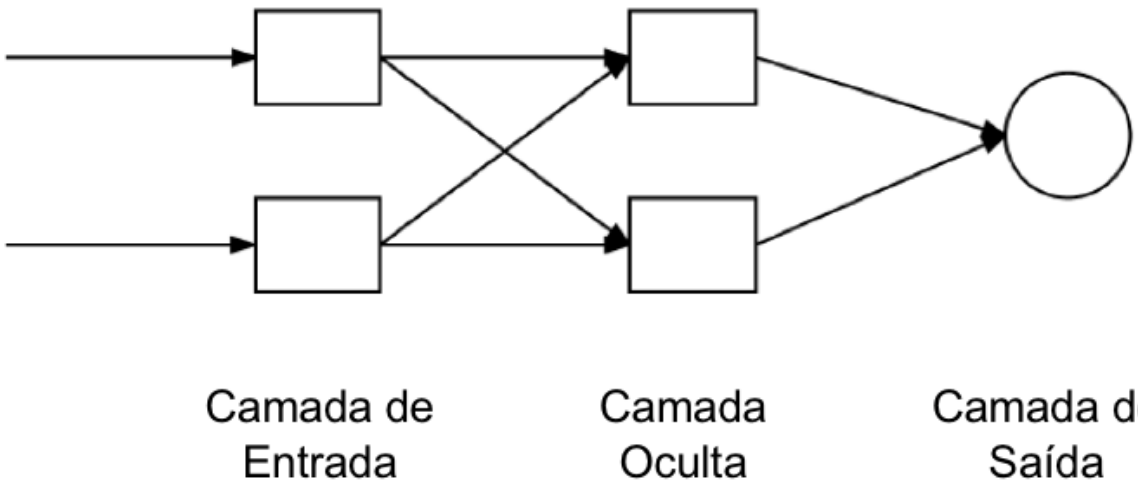
Vamos analisar cada uma dessas fases e compreender como funciona o backpropagation no treinamento nas redes neurais. No próximo capítulo, voltaremos ao script em Python para compreender como é a implementação do algoritmo. Let's begin!

Forward Pass

O propósito do passo para frente é propagar nossas entradas (os dados de entrada) através da rede aplicando uma série de *dot products* (multiplicação entre os vetores) e ativações até chegarmos à camada de saída da rede (ou seja, nossas previsões). Para visualizar esse processo, vamos primeiro considerar a tabela abaixo. Podemos ver que cada entrada X na matriz é 2-dim (2 dimensões), onde cada ponto de dado é representado por dois números. Por exemplo, o primeiro ponto de dado é representado pelo vetor de recursos (0, 0), o segundo ponto de dado por (0, 1), etc. Em seguida, temos nossos valores de saída Y como a coluna da direita. Nossos valores de saída são os rótulos de classe. Dada uma entrada da matriz, nosso objetivo é prever corretamente o valor de saída desejado. Em resumo, X representa as entradas e Y a saída.

X0	X1	Y
0	0	0
0	1	1
1	0	1
1	1	0

Para obter uma precisão de classificação perfeita nesse problema, precisamos de uma rede neural feedforward com pelo menos uma camada oculta. Podemos então começar com uma arquitetura de 2-2-1 conforme a imagem abaixo.



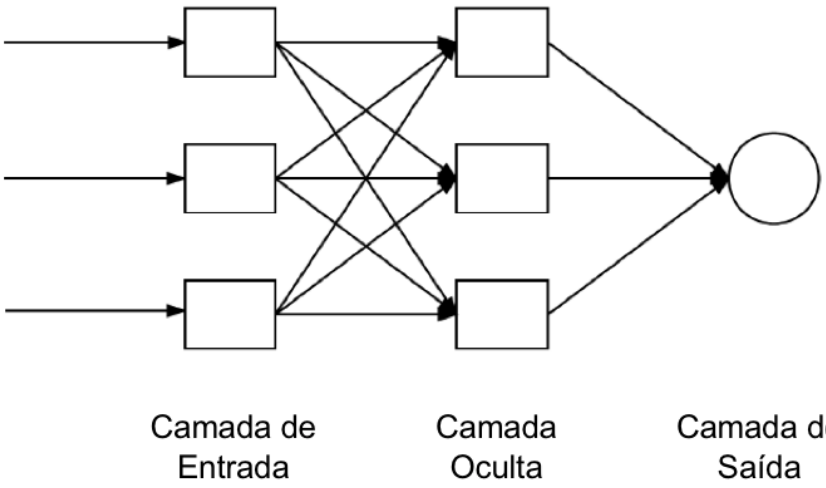
Este é um bom começo, no entanto, estamos esquecendo de incluir o bias. Existem duas maneiras de incluir o bias b em nossa rede. Nós podemos:

1. Usar uma variável separada.
2. Tratar o bias como um parâmetro treinável dentro da matriz, inserindo uma coluna de **1s** nos vetores de recursos.

Inserir uma coluna de 1s no nosso vetor de recursos é feito de forma programática, mas para garantir a didática, vamos atualizar nossa matriz para ver isso explicitamente, conforme tabela abaixo. Como você pode ver, uma coluna de **1s** foi adicionada aos nossos vetores de recursos. Na prática você pode inserir essa coluna em qualquer lugar que desejar, mas normalmente a colocamos como a primeira entrada no vetor de recursos ou a última entrada no vetor de recursos.

X0	X1	X2 (bias)	Y
0	0	1	0
0	1	1	1
1	0	1	1
1	1	1	0

Como nós mudamos o tamanho do nosso vetor de recursos de entrada (normalmente o que é realizado dentro da implementação da rede em si, para que não seja necessário modificar explicitamente a nossa matriz), isso muda nossa arquitetura de rede de 2-2-1 para uma arquitetura 3-3-1, conforme imagem abaixo. Ainda nos referimos a essa arquitetura de rede como 2-2-1, mas quando se trata de implementação, na verdade, é 3-3-1 devido à adição do termo de bias incorporado na matriz.



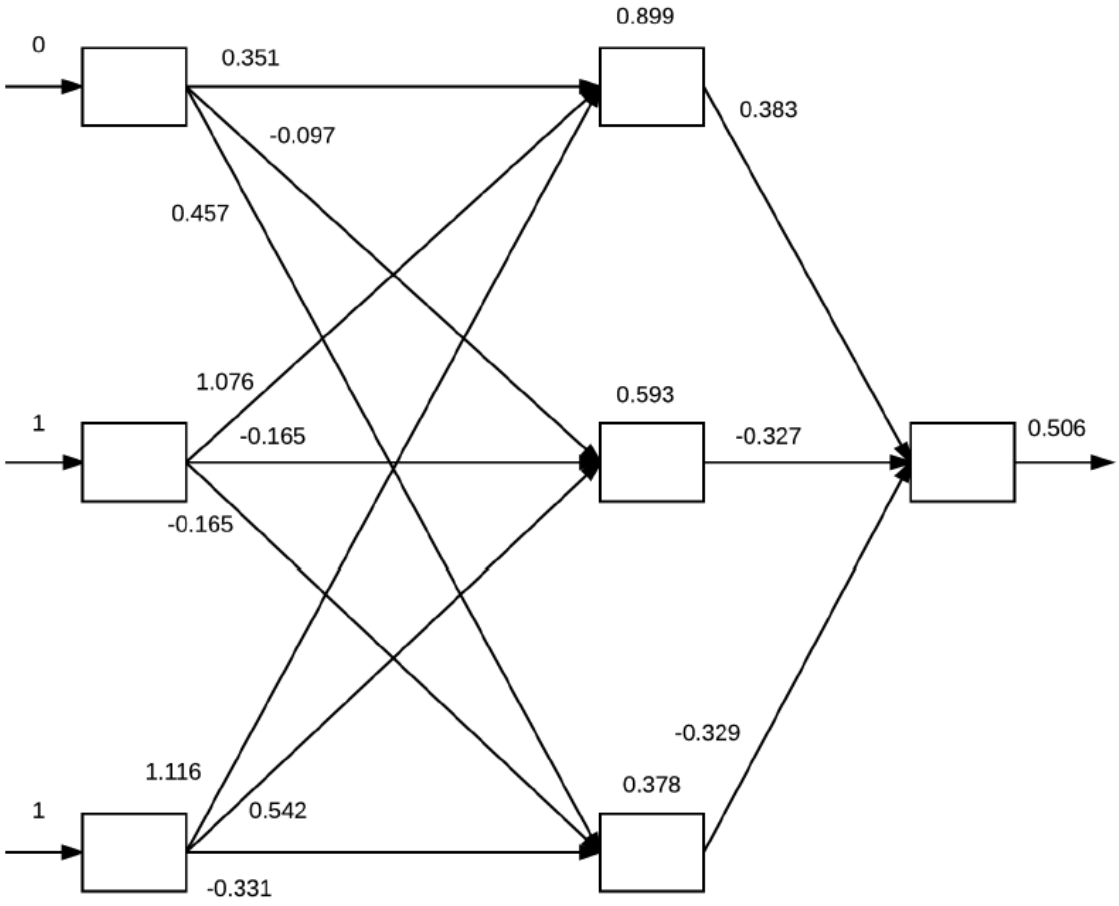
Finalmente, lembre-se de que tanto nossa camada de entrada quanto todas as camadas ocultas exigem um termo de bias. No entanto, a camada de saída final não requer um bias. O bias agora é um parâmetro treinável dentro da matriz de peso, tornando o treinamento mais eficiente e substancialmente mais fácil de implementar. Para ver o forward pass em ação, primeiro inicializamos os pesos em nossa rede, conforme figura abaixo. Observe como cada seta na matriz de peso tem um valor associado a ela – esse é o valor de peso atual para um determinado nó e significa o valor em que uma determinada entrada é amplificada ou diminuída. Este valor de peso será então atualizado durante a fase de backpropgation (lembre-se que ainda estamos no forward pass). Existem várias formas de inicializar o vetor de pesos e isso pode influenciar diretamente no treinamento da rede, como veremos mais abaixo.

Na extrema esquerda da figura abaixo, apresentamos o vetor de recursos (0, 1, 1) e também o valor de saída 1 para a rede, pois depois precisamos calcular os erros de previsão. Aqui podemos ver que 0,1 e 1 foram atribuídos aos três nós de entrada na rede. Para propagar os valores através da rede e obter a classificação final, nós precisamos do *dot product* entre as entradas e os valores de peso, seguido pela aplicação de um função de ativação (neste caso, a função **sigmóide s**). Vamos calcular as entradas para os três nós nas camadas ocultas:

1. s $((0 \times 0.351) + (1 \times 1.076) + (1 \times 1.116)) = 0.899$

2. s $((0 \times 0.097) + (1 \times 0.165) + (1 \times 0.542)) = 0.593$

3. s $((0 \times 0.457) + (1 \times 0.165) + (1 \times 0.331)) = 0.378$



Observando os valores dos nós das camadas ocultas (camadas do meio), podemos ver que os nós foram atualizados para refletir nossa computação. Agora temos nossas entradas para os nós da camada oculta. Para calcular a previsão de saída, uma vez mais usamos o *dot product* seguido por uma ativação sigmóide:

$$s((0.899 \times 0.383) + (0.593 \times -0.327) + (0.378 \times -0.329)) = 0.506$$

A saída da rede é, portanto, 0.506. Podemos aplicar uma função de etapa (step function) para determinar se a saída é a classificação correta ou não:

$$f(saida) = \begin{cases} \mathbf{1} \text{ se saida} > 0 \\ \mathbf{0} \text{ se caso contrário} \end{cases}$$

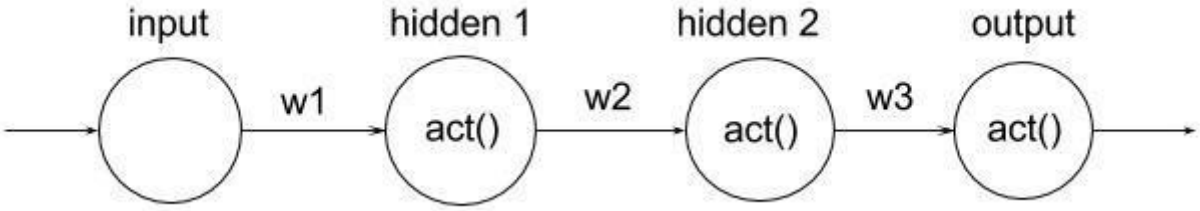
Aplicando a step function com saida = 0.506, vemos que nossa rede prevê 1 que é, de fato, o rótulo de classe correto. No entanto, a nossa rede não está muito confiante neste rótulo de classe. O valor previsto 0.506 está muito próximo do limite da etapa. Idealmente, esta previsão deve ser mais próxima de 0.98 ou 0.99., implicando que a nossa rede realmente aprendeu o padrão no conjunto de dados. Para que nossa rede realmente "aprenda", precisamos aplicar o backpropagation.

Backpropagation

Para qualquer problema de aprendizagem supervisionada, nós selecionamos pesos que fornecem a estimativa ótima de uma função que modela nossos dados de treinamento. Em outras palavras, queremos encontrar um conjunto de pesos W que minimize a saída de J(W), onde J(W) é a função de perda, ou o erro da rede. Nos capítulos anteriores, discutimos o algoritmo de gradiente descendente, em que atualizamos cada peso por alguma redução escalar negativa da derivada do erro em relação a esse peso. Se optarmos por usar gradiente descendente (ou quase qualquer outro algoritmo de otimização convexo), precisamos encontrar as derivadas na forma numérica.

O objetivo do backpropagation é otimizar os pesos para que a rede neural possa aprender a mapear corretamente as entradas para as saídas.

Para outros algoritmos de aprendizado de máquina, como regressão logística ou regressão linear, o cálculo das derivadas é uma aplicação elementar de diferenciação. Isso ocorre porque as saídas desses modelos são apenas as entradas multiplicadas por alguns pesos escolhidos e, no máximo, alimentados por uma única função de ativação (a função sigmóide na regressão logística). O mesmo, no entanto, não pode ser dito para redes neurais. Para demonstrar isso, aqui está um diagrama de uma rede neural de dupla camada:



Como você pode ver, cada neurônio é uma função do anterior conectado a ele. Em outras palavras, se alguém alterasse o valor de w1, os neurônios "hidden 1" e "hidden 2" (e, finalmente, a saída) mudariam. Devido a essa noção de dependências funcionais, podemos formular matematicamente a saída como uma função composta extensiva:

$$output = act(w3 * hidden2)$$
$$hidden2 = act(w2 * hidden1)$$
$$hidden1 = act(w1 * input)$$

ou simplesmente:

$$output = act(w3 * act(w2 * act(w1 * input)))$$

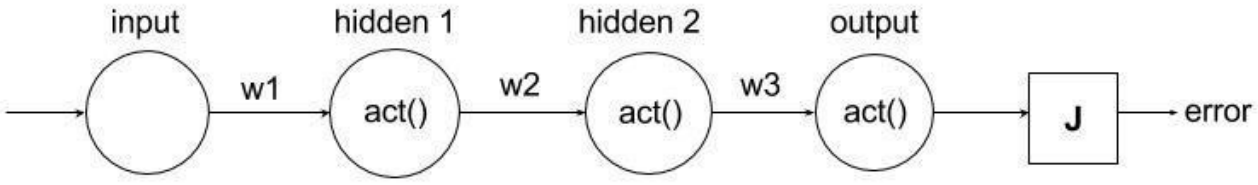
Para aplicar o algoritmo de backpropagation, nossa função de ativação deve ser diferenciável, de modo que possamos calcular a derivada parcial do erro em relação a um dado peso wi,j, loss(E), saída de nó oj e saída de rede j.

$$\frac{\partial E}{\partial w_{i,j}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial net_j} \frac{\partial net_j}{\partial w_{i,j}}$$

Aqui, a saída é uma função composta dos pesos, entradas e função (ou funções) de ativação. É importante perceber que as unidades / nós ocultos são simplesmente cálculos intermediários que, na realidade, podem ser reduzidos a cálculos da camada de entrada. Se fôssemos então tirar a derivada da função com relação a algum peso arbitrário (por exemplo, w1), aplicaríamos iterativamente a regra da cadeia (da qual eu tenho certeza que você se lembra do capítulo anterior). O resultado seria semelhante ao seguinte:

$$\frac{\partial}{\partial w1} output = \frac{\partial}{\partial hidden2} output * \frac{\partial}{\partial hidden1} hidden2 * \frac{\partial}{\partial w1} hidden1$$

Agora, vamos anexar mais uma operação à cauda da nossa rede neural. Esta operação irá calcular e retornar o erro – usando a função de custo – da nossa saída:



Tudo o que fizemos foi adicionar outra dependência funcional; nosso erro é agora uma função da saída e, portanto, uma função da entrada, pesos e função de ativação. Se fôssemos calcular a derivada do erro com qualquer peso arbitrário (novamente, escolheríamos w1), o resultado seria:

$$\frac{\partial error}{\partial w1} = \frac{\partial error}{\partial output} * \frac{\partial output}{\partial hidden2} * \frac{\partial hidden2}{\partial hidden1} * \frac{\partial hidden1}{\partial w1}$$

Cada uma dessas derivações pode ser simplificada, uma vez que escolhemos uma função de ativação e erro, de modo que todo o resultado represente um valor numérico. Nesse ponto, qualquer abstração foi removida e a derivada de erro pode ser usada na descida do gradiente (como discutido anteriormente aqui no livro) para melhorar iterativamente o peso. Calculamos as derivadas de erro w.r.t. para todos os outros pesos na rede e aplicamos gradiente descendente da mesma maneira. Isso é backpropagation – simplesmente o cálculo de derivadas que são alimentadas para um algoritmo de otimização convexa. Chamamos isso de "retropropagação" porque estamos usando o erro de saída para atualizar os pesos, tomando passos iterativos usando a regra da cadeia até que alcancemos o valor de peso ideal.

Depois de compreender o funcionamento do algoritmo backpropagation, você percebe sua simplicidade. Claro, a aritmética/cálculos reais podem ser difíceis, mas esse processo é tratado pelos nossos computadores. Na realidade, o backpropagation é apenas uma aplicação da regra da cadeia (chain rule). Como as redes neurais são estruturas de modelo de aprendizado de máquina multicamadas complicadas, cada peso "contribui" para o erro geral de uma maneira mais complexa e, portanto, as derivadas reais exigem muito esforço para serem produzidas. No entanto, uma vez que passamos pelo cálculo, o backpropagation das redes neurais é equivalente à descida de gradiente típica para regressão logística / linear.

Assim, como regra geral de atualizações de peso, podemos usar a Regra Delta (Delta Rule):

Novo Peso = Peso Antigo – Derivada * Taxa de Aprendizagem

A taxa de aprendizagem (learning rate) é introduzida como uma constante (geralmente muito pequena), a fim de forçar o peso a ser atualizado de forma suave e lenta (para evitar grandes passos e comportamento caótico).

Para validar esta equação:

- Se a Derivada for positiva, isso significa que um aumento no peso aumentará o erro, portanto, o novo peso deverá ser menor.
- Se a Derivada é negativa, isso significa que um aumento no peso diminuirá o erro, portanto, precisamos aumentar os pesos.
- Se a Derivada é 0, significa que estamos em um mínimo estável. Assim, nenhuma atualização nos pesos é necessária -> chegamos a um estado estável.

Existem vários métodos de atualização de peso. Esses métodos são frequentemente chamados de *otimizadores*. A regra delta é a mais simples e intuitiva, no entanto, possui várias desvantagens. Confira nas referências ao final do capítulo, um excelente artigo sobre otimizadores.

Como atualizamos os pesos com uma pequena etapa delta de cada vez, serão necessárias várias iterações para ocorrer o aprendizado. Na rede neural, após cada iteração, a força de descida do gradiente atualiza os pesos para um valor cada vez menor da função de perda global. A atualização de peso na rede neural é guiada pela força do gradiente descendente sobre o erro.

Quantas iterações são necessárias para convergir (ou seja, alcançar uma função de perda mínima global)? Isso vai depender de diversos fatores:

- Depende de quão forte é a taxa de aprendizado que estamos aplicando. Alta taxa de aprendizado significa aprendizado mais rápido, mas com maior chance de instabilidade.
- Depende também dos hyperparâmetros da rede (quantas camadas, quão complexas são as funções não-lineares, etc..). Quanto mais variáveis, mais leva tempo para convergir, mas a precisão tende a ser maior.
- Depende do uso do método de otimização, pois algumas regras de atualização de peso são comprovadamente mais rápidas do que outras.
- Depende da inicialização aleatória da rede. Talvez com alguma sorte você inicie a rede com pesos quase ideais e esteja a apenas um passo da solução ideal. Mas o contrário também pode ocorrer.
- Depende da qualidade do conjunto de treinamento. Se a entrada e a saída não tiverem correlação entre si, a rede neural não fará mágica e não poderá aprender uma correlação aleatória.

Ou seja, treinar uma rede neural não é tarefa simples. Imagine agora treinar uma rede profunda, com várias camadas intermediárias e milhões ou mesmo bilhões de pontos de dados e você compreende o quão trabalhoso isso pode ser e quantas decisões devem ser tomadas pelo Cientista de Dados ou Engenheiro de IA. E aprender a trabalhar de forma profissional, requer tempo, dedicação e preparo e melhor ainda se isso puder ser 100% em português para acelerar seu aprendizado. Construir aplicações de IA é uma habilidade com demanda cada vez maior no mercado.

Pensando nisso, a Data Science Academy oferece um programa completo, onde esses e vários outros conceitos são estudados em detalhes e com várias aplicações práticas, usando TensorFlow. A Formação Inteligência Artificial é composta de 9 cursos, tudo 100% online e 100% em português, que aliam teoria e prática na medida certa, com aplicações reais de Inteligência Artificial. Confira o programa completo dos cursos: Formação Inteligência Artificial. Várias empresas em todo Brasil já estão treinando seus profissionais conosco! Venha fazer parte da revolução da IA.

Agora que você já compreende como funciona o backpropagation, podemos retornar ao código Python e ver tudo isso funcionando na prática. Mas isso é assunto para o próximo capítulo!

Referências:

Formação Inteligência Artificial

Machine Learning

Dot Product

How the backpropagation algorithm works

Delta Rule

An overview of gradient descent optimization algorithms

Neural Networks & The Backpropagation Algorithm, Explained

Derivada

Machine Learning

The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition

Gradient Descent For Machine Learning

Pattern Recognition and Machine Learning

Understanding Activation Functions in Neural Networks

Redes Neurais, princípios e práticas

An overview of gradient descent optimization algorithms

Optimization: Stochastic Gradient Descent

Gradient Descent vs Stochastic Gradient Descent vs Mini-Batch Learning

CAPÍTULO 16

Algoritmo Backpropagation em Python

Depois de compreender como funciona o backpropagation, podemos agora entender o código usado em alguns capítulos anteriores para implementar o algoritmo (o qual vamos reproduzir aqui). O arquivo com o código completo pode ser encontrado no repositório do livro no Github.

Em nosso código nós temos os métodos **update_mini_batch** e **backprop** da classe Network. Em particular, o método **update_mini_batch** atualiza os pesos e bias da rede calculando o gradiente para o mini_batch atual de exemplos (dados) de treinamento:

```
def update_mini_batch(self, mini_batch, eta):
    """Atualiza os pesos e bias da rede aplicando
     a descida do gradiente usando backpropagation para um único mini lote.
     O `mini_batch` é uma lista de tuplas `(x, y)`, e `eta` é a taxa de aprendizado."""

    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]

    for x, y in mini_batch:
        delta_nabla_b, delta_nabla_w = self.backprop(x, y)
        nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
        nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]

    self.weights = [w-(eta/len(mini_batch))*nw for w, nw in zip(self.weights, nabla_w)]
    self.biases = [b-(eta/len(mini_batch))*nb for b, nb in zip(self.biases, nabla_b)]
```

A maior parte do trabalho é feita pela linha:

delta_nabla_b, delta_nabla_w = self.backprop (x, y)

que usa o método backprop para descobrir as derivadas parciais ∂Cx / ∂blj e ∂Cx / ∂wljk. Isso invoca o algoritmo de backpropagation, que é uma maneira rápida de calcular o gradiente da função de custo. Portanto, update_mini_batch funciona simplesmente calculando esses gradientes para cada exemplo de treinamento no mini_batch e, em seguida, atualizando self.weights e self.biases adequadamente. Há uma pequena mudança – usamos uma abordagem ligeiramente diferente para indexar as camadas. Essa alteração é feita para aproveitar um recurso do Python, ou seja, o uso de índices de lista negativa para contar para trás a partir do final de uma lista, por exemplo, lst[-3] é a terceira última entrada em uma lista chamada lst. O código para backprop está abaixo, junto com algumas funções auxiliares, que são usadas para calcular a função σ, a derivada σ′ e a derivada da função de custo. Com essas inclusões, você deve ser capaz de entender o código de maneira independente:

```
def backprop(self, x, y):
    """Retorna uma tupla `(nabla_b, nabla_w)` representando o
     gradiente para a função de custo C_x. `nabla_b` e
     `nabla_w` são listas de camadas de matrizes numpy, semelhantes
     a `self.biases` e `self.weights`."""

    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]

    # Feedforward
    activation = x

    # Lista para armazenar todas as ativações, camada por camada
    activations = [x]

    # Lista para armazenar todos os vetores z, camada por camada
    zs = []

    for b, w in zip(self.biases, self.weights):
        z = np.dot(w, activation)+b
        zs.append(z)
        activation = sigmoid(z)
        activations.append(activation)

    # Backward pass
    delta = self.cost_derivative(activations[-1], y) * sigmoid_prime(zs[-1])
    nabla_b[-1] = delta
    nabla_w[-1] = np.dot(delta, activations[-2].transpose())

    # Aqui, l = 1 significa a última camada de neurônios, l = 2 é a
    # segunda e assim por diante.
    for l in range(2, self.num_layers):
        z = zs[-l]
        sp = sigmoid_prime(z)
        delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
        nabla_b[-l] = delta
        nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
    return (nabla_b, nabla_w)
```

Observe o método backprop. Começamos inicalizando as matrizes de pesos (nabla_w) e bias (nabla_b) com zeros. Essas matrizes serão alimentadas com valores durante o processo de treinamento. Isso é o que a rede neural artificial efetivamente aprende. Depois de inicializar alguns objetos, temos um loop for para cada valor de b e w (que a esta altura você já sabe se trata de bias e pesos, respectivamente). Neste loop, usamos a função np.dot do Numpy para a multiplicação entre matrizes e adição do bias, colocamos o resultado na lista z e fazemos uma chamada à função de ativação Sigmóide. Ao final deste loop, teremos a lista com todas as ativações e finalizamos a passada para a frente.

Na passada para trás (Backward Pass) calculamos as derivadas e fazemos as multiplicações de matrizes mais uma vez (o funcionamento de redes neurais artificiais é baseado em um conceito elementar da Álgebra Linear, a multiplicação de matrizes). Repare que chamamos o método Transpose() para gerar a transposta da matriz e assim ajustar as dimensões antes de efetuar os cálculo. Por fim, retornamos bias e pesos.

Em que sentido backpropagation é um algoritmo rápido?

Para responder a essa pergunta, vamos considerar outra abordagem para calcular o gradiente. Imagine que é o início da pesquisa de redes neurais. Talvez seja a década de 1950 ou 1960, e você é a primeira pessoa no mundo a pensar em usar gradiente descendente para o aprendizado! Mas, para que a ideia funcione, você precisa de uma maneira de calcular o gradiente da função de custo. Você volta ao seu conhecimento de cálculo e decide se pode usar a regra da cadeia (chain rule) para calcular o gradiente. Mas depois de brincar um pouco, a álgebra parece complicada e você fica desanimado. Então você tenta encontrar outra abordagem. Você decide considerar o custo como uma função apenas dos pesos C = C(w) (voltaremos ao bias em um momento). Você numera os pesos w1, w2,… e deseja computar ∂C / ∂wj para um peso específico wj. Uma maneira óbvia de fazer isso é usar a aproximação

$$\frac{\partial C}{\partial w_j} \approx \frac{C(w + \epsilon e_j) - C(w)}{\epsilon}$$

onde $\epsilon > 0$ é um pequeno número positivo e ej é o vetor unitário na direção j. Em outras palavras, podemos estimar ∂C / ∂wj calculando o custo C para dois valores ligeiramente diferentes de wj e, em seguida, aplicando a equação. A mesma ideia nos permitirá calcular as derivadas parciais ∂C / ∂b em relação aos vieses (bias).

Essa abordagem parece muito promissora. É simples conceitualmente e extremamente fácil de implementar, usando apenas algumas linhas de código. Certamente, parece muito mais promissor do que a ideia de usar a regra da cadeia para calcular o gradiente!

Infelizmente, embora essa abordagem pareça promissora, quando você implementa o código, ele fica extremamente lento. Para entender porque, imagine que temos um milhão de pesos em nossa rede. Então, para cada peso distinto wj, precisamos computar C (w + ϵej) para calcular ∂C / ∂wj. Isso significa que, para calcular o gradiente, precisamos computar a função de custo um milhão de vezes diferentes, exigindo um milhão de passos para frente pela rede (por exemplo, treinamento). Precisamos calcular C(w) também, em um total de um milhão de vezes e em uma única passada pela rede.

O que há de inteligente no backpropagation é que ele nos permite calcular simultaneamente todas as derivadas parciais ∂C / ∂wj usando apenas uma passagem direta pela rede, seguida por uma passagem para trás pela rede. Grosso modo, o custo computacional do passe para trás é quase o mesmo que o do forward. Isso deve ser plausível, mas requer algumas análises para fazer uma declaração cuidadosa. É plausível porque o custo computacional dominante no passe para frente é multiplicado pelas matrizes de peso, enquanto no passo para trás é multiplicado pelas transpostas das matrizes de peso. Obviamente, essas operações têm um custo computacional similar. E assim, o custo total da retropropagação (backpropagation) é aproximadamente o mesmo que fazer apenas duas passagens pela rede. Compare isso com o milhão e um passe para frente que precisávamos para a abordagem que descrevi anteriormente. E assim, embora a retropropagação pareça superficialmente mais complexa do que a abordagem anterior, é na verdade muito, muito mais rápida.

Essa aceleração foi amplamente apreciada em 1986 e expandiu enormemente a gama de problemas que as redes neurais poderiam resolver. Isso, por sua vez, causou uma onda de pessoas usando redes neurais. Claro, a retropropagação não é uma panacéia. Mesmo no final da década de 1980, as pessoas enfrentavam limites, especialmente quando tentavam usar a retropropagação para treinar redes neurais profundas, ou seja, redes com muitas camadas ocultas. Mais adiante, no livro, veremos como os computadores modernos e algumas novas ideias inteligentes tornam possível usar a retropropagação para treinar redes neurais bem profundas.

Seu trabalho agora é estudar e compreender cada linha de código usada em nossa rede de amostra. Esse código é bem simples e o objetivo é mostrar a você como as coisas funcionam programaticamente. Ainda vamos treinar nossa rede, avaliar seu desempenho, otimizar algumas operações e compreender outros conceitos básicos. Temos muito mais vindo por aí! Até o próximo capítulo!

Referências:

Formação Inteligência Artificial

Machine Learning

Dot Product

How the backpropagation algorithm works

Delta Rule

An overview of gradient descent optimization algorithms

Neural Networks & The Backpropagation Algorithm, Explained

Derivada

Machine Learning

The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition

Gradient Descent For Machine Learning

Pattern Recognition and Machine Learning

Understanding Activation Functions in Neural Networks

Redes Neurais, princípios e práticas

An overview of gradient descent optimization algorithms

Optimization: Stochastic Gradient Descent

Gradient Descent vs Stochastic Gradient Descent vs Mini-Batch Learning

CAPÍTULO 17

Cross-Entropy Cost Function

Quando um jogador de tênis está aprendendo a praticar o esporte, ele geralmente passa a maior parte do tempo desenvolvendo o movimento do corpo. Apenas gradualmente ele desenvolve as tacadas, aprende a movimentar a bola com precisão para a quadra adversária e com isso vai construindo sua técnica, que se aprimora à medida que ele pratica. De maneira semelhante, até agora nos concentramos em entender o algoritmo de retropropagação (backpropagation), a base para aprender a maioria das atividades em redes neurais. A partir de agora, estudaremos um conjunto de técnicas que podem ser usadas para melhorar nossa implementação do backpropagation e, assim, melhorar a maneira como nossas redes aprendem.

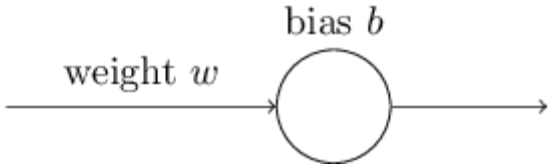
As técnicas que desenvolveremos incluem: uma melhor escolha de função de custo, conhecida como função de custo de entropia cruzada (ou Cross-Entropy Cost Function); quatro métodos de "regularização" (regularização de L1 e L2, dropout e expansão artificial dos dados de treinamento), que melhoram nossas redes para generalizar além dos dados de treinamento; um método melhor para inicializar os pesos na rede; e um conjunto de heurísticas para ajudar a escolher bons hyperparâmetros para a rede. Também vamos analisar várias outras técnicas com menos profundidade. As discussões são em grande parte independentes umas das outras e, portanto, você pode avançar se quiser. Também implementaremos muitas das técnicas em nosso código e usaremos para melhorar os resultados obtidos no problema de classificação de dígitos manuscritos estudado nos capítulos anteriores.

Naturalmente, estamos cobrindo apenas algumas das muitas técnicas que foram desenvolvidas para uso em redes neurais. A filosofia é que o melhor acesso à multiplicidade de técnicas disponíveis é o estudo aprofundado de algumas das mais importantes. Dominar essas técnicas importantes não é apenas útil por si só, mas também irá aprofundar sua compreensão sobre quais problemas podem surgir quando você usa redes neurais. Isso deixará você bem preparado para aprender rapidamente outras técnicas, conforme necessário.

A Função de Custo

A maioria de nós acha desagradável estar errado. Logo depois de começar a aprender piano, minha filha fez sua primeira apresentação diante de uma platéia. Ela estava nervosa e começou a tocar a peça com uma oitava muito baixa. Ela ficou confusa e não pôde continuar até que alguém apontasse o erro. Ela ficou muito envergonhada. Ainda que desagradável, também aprendemos rapidamente quando estamos decididamente errados. Você pode apostar que a próxima vez que ela se apresentou diante de uma platéia, ela começou na oitava correta! Em contraste, aprendemos mais lentamente quando nossos erros são menos bem definidos.

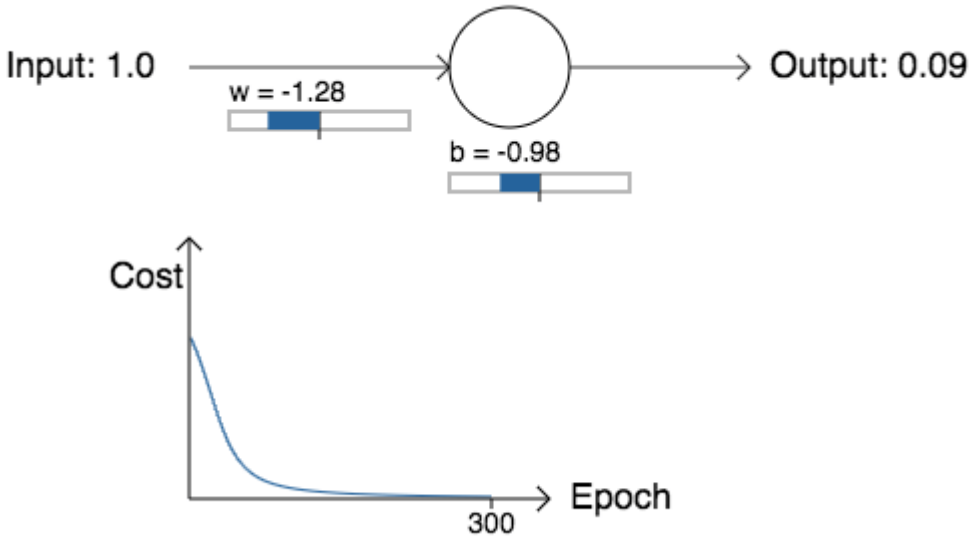
Idealmente, esperamos que nossas redes neurais aprendam rapidamente com seus erros. Mas é isso que acontece na prática? Para responder a essa pergunta, vamos dar uma olhada em um exemplo simples. O exemplo envolve um neurônio com apenas uma entrada:



Nós vamos treinar esse neurônio para fazer algo ridiculamente fácil: obter a entrada 1 e gerar a saída 0. Claro, essa é uma tarefa tão trivial que poderíamos facilmente descobrir um peso apropriado e um viés (bias) de forma manual, sem usar um algoritmo de aprendizado. No entanto, vai nos ajudar a compreender melhor o processo de usar gradiente descendente para tentar aprender um peso e viés. Então, vamos dar uma olhada em como o neurônio aprende.

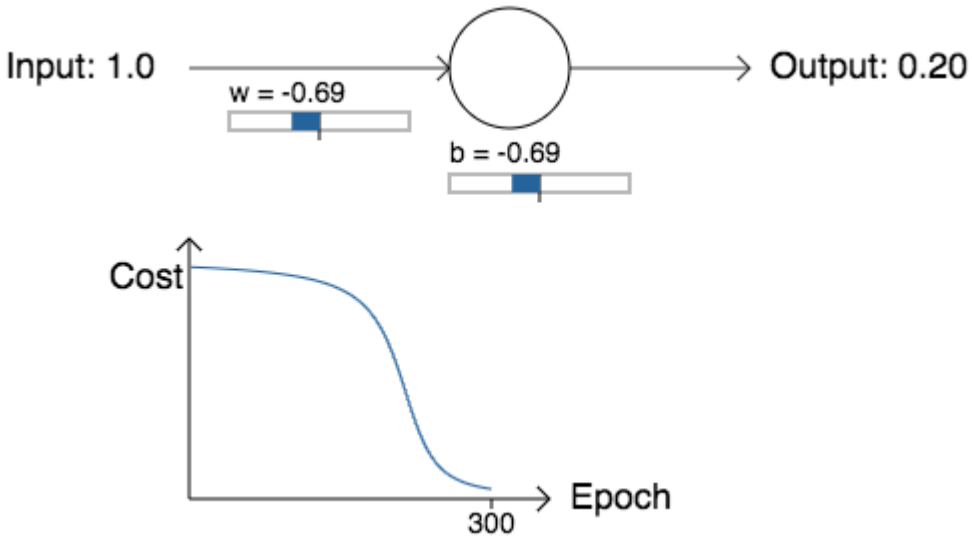
Para tornar as coisas definitivas, escolhemos o peso inicial como 0.6 e o viés inicial como 0.9. Estas são escolhas genéricas usadas como um lugar para começar a aprender, eu não as escolhi para serem especiais de alguma forma. A saída inicial do neurônio é 0.82, então um pouco de aprendizado será necessário antes que nosso neurônio se aproxime da saída desejada 0,0.

No gráfico abaixo, podemos ver como o neurônio aprende uma saída muito mais próxima de 0.0. Durante o treinamento, o modelo está realmente computando o gradiente, e usando o gradiente para atualizar o peso e o viés, e exibir o resultado. A taxa de aprendizado é $\eta = 0.15$, o que acaba sendo lento o suficiente para que possamos acompanhar o que está acontecendo, mas rápido o suficiente para que possamos obter um aprendizado substancial em apenas alguns segundos. O custo é a função de custo quadrático, C, apresentada nos capítulos anteriores. Vou lembrá-lo da forma exata da função de custo em breve.



Como você pode ver, o neurônio aprende um peso e um viés que diminui o custo e dá uma saída do neurônio de cerca de 0.09 (Epoch, ou Época em português, é o número de passadas que nosso modelo faz pelos dados. A cada passada, os pesos são atualizados, o aprendizado ocorre e o custo, ou a taxa de erros, diminui). Isso não é exatamente o resultado desejado, 0.0, mas é muito bom.

Suponha, no entanto, que, em vez disso, escolhamos o peso inicial e o viés inicial como 2.0. Nesse caso, a saída inicial é 0.98, o que é muito ruim. Vamos ver como o neurônio aprende a gerar 0 neste caso:



Embora este exemplo use a mesma taxa de aprendizado ($\eta = 0.15$), podemos ver que a aprendizagem começa muito mais devagar. De fato, nas primeiras 150 épocas de aprendizado, os pesos e vieses não mudam muito. Então o aprendizado entra em ação e, como em nosso primeiro exemplo, a saída do neurônio se aproxima rapidamente de 0.0.

Esse comportamento é estranho quando comparado ao aprendizado humano. Como eu disse no começo deste capítulo, muitas vezes aprendemos mais rápido quando estamos muito errados sobre algo. Mas acabamos de ver que nosso neurônio artificial tem muita dificuldade em aprender quando está muito errado – muito mais dificuldade do que quando está apenas um pouco errado. Além do mais, verifica-se que esse comportamento ocorre não apenas neste exemplo, mas em redes mais gerais. Por que aprender tão devagar? E podemos encontrar uma maneira de evitar essa desaceleração?

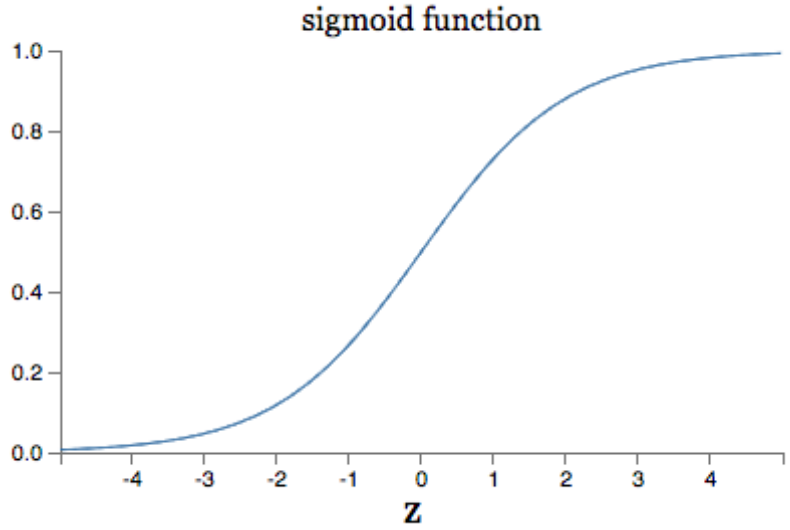
Para entender a origem do problema, considere que nosso neurônio aprende mudando o peso e o viés a uma taxa determinada pelas derivadas parciais da função custo, $\partial C/\partial w$ e $\partial C/\partial b$. Então, dizer "aprender é lento" é realmente o mesmo que dizer que essas derivadas parciais são pequenas. O desafio é entender por que eles são pequenas. Para entender isso, vamos calcular as derivadas parciais. Lembre-se de que estamos usando a função de custo quadrático, que é dada por:

$$C = \frac{(y-a)^2}{2}$$

onde **a** é a saída do neurônio quando a entrada de treinamento $x = 1$ é usada, e $y = 0$ é a saída desejada correspondente. Para escrever isso mais explicitamente em termos de peso e viés, lembre-se que $a = \sigma(z)$, onde $z = wx + b$. Usando a regra da cadeia para diferenciar em relação ao peso e viés, obtemos:

$$\frac{\partial C}{\partial w} = (a - y)\sigma'(z)x = a\sigma'(z)$$
$$\frac{\partial C}{\partial b} = (a - y)\sigma'(z) = a\sigma'(z),$$

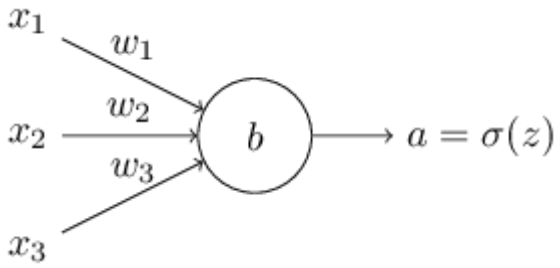
onde substitui x = 1 e y = 0. Para entender o comportamento dessas expressões, vamos olhar mais de perto o termo σ ′ (z) no lado direito. Lembre-se da forma da função σ:



Podemos ver neste gráfico que quando a saída do neurônio é próxima de 1, a curva fica muito plana, e então σ ′ (z) fica muito pequeno. As equações acima então nos dizem que ∂C/∂w e ∂C/∂b ficam muito pequenos. Esta é a origem da desaceleração da aprendizagem. Além do mais, como veremos mais adiante, a desaceleração do aprendizado ocorre basicamente pelo mesmo motivo em redes neurais mais genéricas, não apenas neste exemplo simples.

A Função de Custo de Entropia Cruzada

Como podemos abordar a desaceleração da aprendizagem? Acontece que podemos resolver o problema substituindo o custo quadrático por uma função de custo diferente, conhecida como entropia cruzada. Para entender a entropia cruzada, vamos nos afastar um pouco do nosso modelo super-simples. Vamos supor que estamos tentando treinar um neurônio com diversas variáveis de entrada, x1, x2,..., pesos correspondentes w1, w2,... e um viés, b:



A saída do neurônio é, naturalmente, **a = σ(z)**, onde **z = ∑jwjxj + b** é a soma ponderada das entradas. Nós definimos a função de custo de entropia cruzada para este neurônio assim:

$$C = -\frac{1}{n} \sum_x [y \ln a + (1 - y) \ln(1 - a)]$$

onde **n** é o número total de itens de dados de treinamento, a soma é sobre todas as entradas de treinamento **x,** e **y** é a saída desejada correspondente. Não é óbvio que a expressão anterior resolva o problema de desaceleração do aprendizado. De fato, francamente, nem é óbvio que faz sentido chamar isso de uma função de custo! Antes de abordar a desaceleração da aprendizagem, vamos ver em que sentido a entropia cruzada pode ser interpretada como uma função de custo.

Duas propriedades em particular tornam razoável interpretar a entropia cruzada como uma função de custo. Primeiro, não é negativo, isto é, C > 0. Para visualizar isso, observe na fórmula anterior que: (a) todos os termos individuais na soma são negativos, já que ambos os logaritmos são de números no intervalo de 0 a 1; e (b) há um sinal de menos na frente da soma.

Segundo, se a saída real do neurônio estiver próxima da saída desejada para todas as entradas de treinamento x, então a entropia cruzada será próxima de zero. Para ver isso, suponha, por exemplo, que y = 0 e a ≈ 0 para alguma entrada x. Este é um caso quando o neurônio está fazendo um bom trabalho nessa entrada. Vemos que o primeiro termo (na fórmula acima) para o custo, desaparece, desde que y = 0, enquanto o segundo termo é apenas −ln (1 − a) ≈ 0. Uma análise semelhante é válida quando y = 1 e a ≈ 1. E assim, a contribuição para o custo será baixa, desde que a saída real esteja próxima da saída desejada.

Em suma, a entropia cruzada é positiva e tende a zero, à medida que o neurônio melhora a computação da saída desejada, y, para todas as entradas de treinamento, x.

Essas são as duas propriedades que esperamos intuitivamente para uma função de custo. De fato, ambas as propriedades também são satisfeitas pelo custo quadrático. Portanto, isso é uma boa notícia para a entropia cruzada. Mas a função custo de entropia cruzada tem o benefício de que, ao contrário do custo quadrático, evita o problema de desaceleração do aprendizado. Para ver isso, vamos calcular a derivada parcial do custo de entropia cruzada em relação aos pesos. Substituímos a = σ (z) na fórmula acima e aplicamos a regra da cadeia duas vezes, obtendo:

$$\frac{\partial C}{\partial w_j} = -\frac{1}{n}\sum_x \left(\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)}\right)\frac{\partial \sigma}{\partial w_j}$$
$$= -\frac{1}{n}\sum_x \left(\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)}\right)\sigma'(z)x_j.$$

Colocando tudo em um denominador comum e simplificando, isso se torna:

$$\frac{\partial C}{\partial w_j} = \frac{1}{n}\sum_x \frac{\sigma'(z)x_j}{\sigma(z)(1-\sigma(z))}(\sigma(z)-y)$$

Usando a definição da função sigmóide, σ (z) = 1 / (1 + ez), e um pouco de álgebra, podemos mostrar que σ (z) = σ (z) (1 − σ (z)). Vemos que os termos σ′ (z) e σ (z) (1 − σ (z)) se cancelam na equação acima, e simplificando torna-se:

$$\frac{\partial C}{\partial w_j} = \frac{1}{n}\sum_x x_j(\sigma(z)-y)$$

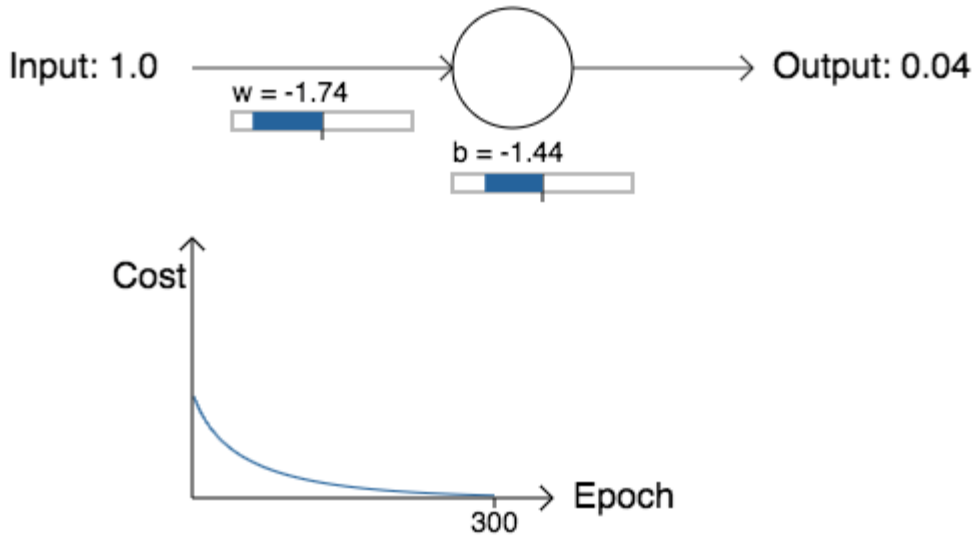
Esta é uma bela expressão. Ela nos diz que a taxa na qual o peso aprende é controlada por σ (z) −y, ou seja, pelo erro na saída. Quanto maior o erro, mais rápido o neurônio aprenderá. Isso é exatamente o que nós esperamos intuitivamente. Em particular, evita a lentidão de aprendizado causada pelo termo σ′ (z) na equação análoga para o custo quadrático. Quando usamos a entropia cruzada, o termo σ′ (z) é cancelado e não precisamos mais nos preocupar em ser pequeno. Este cancelamento é o milagre especial assegurado pela função de custo de entropia cruzada. Na verdade, não é realmente um milagre. Como veremos mais adiante, a entropia cruzada foi especialmente escolhida por ter apenas essa propriedade.

De maneira semelhante, podemos calcular a derivada parcial para o viés. Eu não vou passar por todos os detalhes novamente, mas você pode facilmente verificar que:

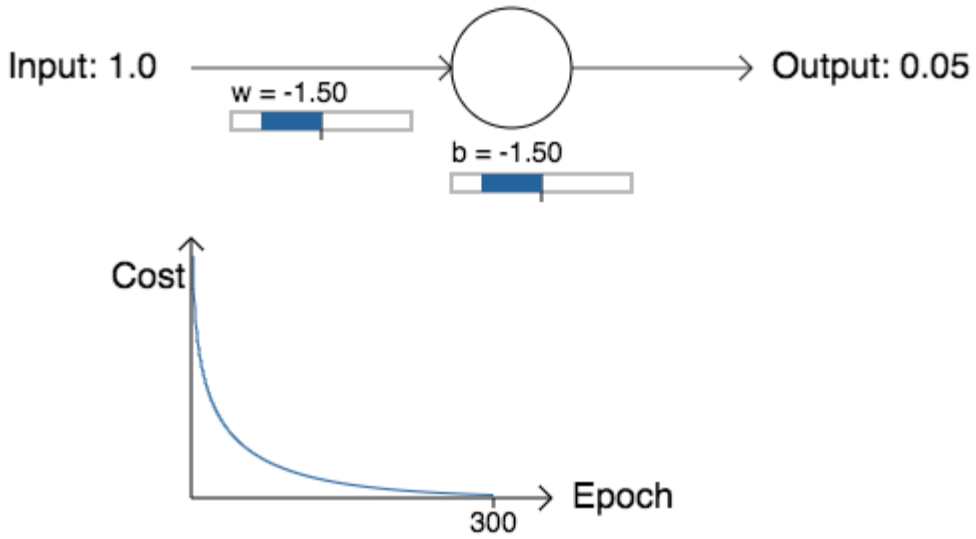
$$\frac{\partial C}{\partial b} = \frac{1}{n}\sum_x (\sigma(z)-y)$$

Novamente, isso evita a lentidão de aprendizado causada pelo termo σ′ (z) na equação análoga para o custo quadrático.

Agora vamos retornar ao exemplo do início deste capítulo, e explorar o que acontece quando usamos a entropia cruzada em vez do custo quadrático. Para nos reorientarmos, começaremos com o caso em que o custo quadrático foi bom, com peso inicial de 0.6 e viés inicial de 0.9. Veja o que acontece quando substituímos o custo quadrático pela entropia cruzada:



Como era de se esperar, o neurônio aprende perfeitamente bem neste caso, assim como fez anteriormente. E agora vamos olhar para o caso em que nosso neurônio ficou preso antes, com o peso e o viés ambos começando em 2.0:



Sucesso! Desta vez, o neurônio aprendeu rapidamente, exatamente como esperávamos. Se você observar atentamente, pode ver que a inclinação da curva de custo era muito mais íngreme inicialmente do que a região plana inicial na curva correspondente para o custo quadrático. É essa inclinação que a entropia cruzada nos ajuda a resolver, impedindo-nos de ficar presos exatamente quando esperamos que nosso neurônio aprenda mais depressa, ou seja, quando o neurônio começa errado.

Eu não disse qual taxa de aprendizado foi usada nos exemplos que acabei de ilustrar. Anteriormente, com o custo quadrático, usamos $\eta = 0.15$. Deveríamos ter usado a mesma taxa de aprendizado nos novos exemplos? De fato, com a mudança na função de custo, não é possível dizer precisamente o que significa usar a "mesma" taxa de aprendizado; é uma comparação de maçãs e laranjas. Para ambas as funções de custo, simplesmente experimentei encontrar uma taxa de aprendizado que possibilitasse ver o que está acontecendo. Se você ainda estiver curioso, aqui está o resumo: usei $\eta = 0.005$ nos exemplos que acabei de fornecer.

Você pode contestar que a mudança na taxa de aprendizado torna os gráficos acima sem sentido. Quem se importa com a rapidez com que o neurônio aprende, quando a nossa escolha de taxa de aprendizado foi arbitrária, para começar ?! Mas essa objeção não procede. O ponto dos gráficos não é sobre a velocidade absoluta de aprendizagem. É sobre como a velocidade do aprendizado muda. Em particular, quando usamos o custo quadrático, a aprendizagem é mais lenta quando o neurônio está inequivocamente errado do que é mais tarde durante o treinamento, à medida que o neurônio se aproxima da saída correta; enquanto o aprendizado de entropia cruzada é mais rápido quando o neurônio está inequivocamente errado. Essas declarações não dependem de como a taxa de aprendizado é definida.

Estamos estudando a entropia cruzada para um único neurônio. No entanto, é fácil generalizar a entropia cruzada para redes multicamadas de muitos neurônios. Em particular, suponha que $y = y1, y2,\ldots$ são os valores desejados nos neurônios de saída, ou seja, os neurônios na camada final, enquanto $aL1, aL2,\ldots$ são os valores reais de saída. Então nós definimos a entropia cruzada por:

$$C = -\frac{1}{n} \sum_x \sum_j \left[y_j \ln a_j^L + (1 - y_j) \ln(1 - a_j^L) \right]$$

Isso é o mesmo que nossa expressão anterior, exceto que agora nós temos o $\sum j$ somando todos os neurônios de saída. Não vou explicitamente trabalhar com uma derivação, mas deve ser plausível que o uso da expressão anterior evite uma desaceleração na aprendizagem em muitas redes de neurônios.

A propósito, estou usando o termo "entropia cruzada" de uma maneira que confundiu alguns dos primeiros leitores, já que parece superficialmente entrar em conflito com outras fontes. Em particular, é comum definir a entropia cruzada para duas distribuições de probabilidade, pj e qj, como $\sum jpjlnqj$. Esta definição pode ser conectada a fórmula da entropia para um neurônio mostrada anteriormente, se tratarmos um único neurônio sigmóide como saída de uma distribuição de probabilidade que consiste na ativação a do neurônio ae seu complemento $1 - a$.

No entanto, quando temos muitos neurônios sigmoides na camada final, o vetor aLj de ativações não costuma formar uma distribuição de probabilidade. Como resultado, uma definição como $\sum jpjlnqj$ não faz sentido, já que não estamos trabalhando com distribuições de probabilidade. Em vez disso, você pode pensar na fórmula da entropia para múltiplos neurônios como um conjunto somado de entropias cruzadas por neurônio, com a ativação de cada neurônio sendo interpretada como parte de uma distribuição de probabilidade de dois elementos. Sim, eu sei que isso não é simples.

Nesse sentido, a fórmula da entropia para múltiplos neurônios é uma generalização da entropia cruzada para distribuições de probabilidade.

Quando devemos usar a entropia cruzada em vez do custo quadrático? De fato, a entropia cruzada é quase sempre a melhor escolha, desde que os neurônios de saída sejam neurônios sigmóides. Para entender por que, considere que, quando estamos configurando a rede, normalmente inicializamos os pesos e vieses usando algum tipo de aleatoriedade. Pode acontecer que essas escolhas iniciais resultem na rede sendo decisivamente errada para alguma entrada de treinamento – isto é, um neurônio de saída terá saturado próximo de 1, quando deveria ser 0, ou vice-versa. Se estamos usando o custo quadrático que irá desacelerar a aprendizagem, ele não vai parar de aprender completamente, já que os pesos continuarão aprendendo com outras entradas de treinamento, mas é obviamente indesejável.

Construir aplicações de IA é uma habilidade com demanda cada vez maior no mercado.

Pensando nisso, a Data Science Academy oferece um programa completo, onde esses e vários outros conceitos são estudados em detalhes e com várias aplicações práticas, usando TensorFlow. A Formação Inteligência Artificial é composta de 9 cursos, tudo 100% online e 100% em português, que aliam teoria e prática na medida certa, com aplicações reais de Inteligência Artificial. Confira o programa completo dos cursos: Formação Inteligência Artificial. Várias empresas em todo Brasil já estão treinando seus profissionais conosco! Venha fazer parte da revolução da IA.

Até o próximo capítulo!

Referências:

Formação Inteligência Artificial

Dot Product

Neural Networks & The Backpropagation Algorithm, Explained

Derivada

Machine Learning

The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition

Gradient Descent For Machine Learning

Pattern Recognition and Machine Learning

Understanding Activation Functions in Neural Networks

Redes Neurais, princípios e práticas

CAPÍTULO 18

Entropia Cruzada para Quantificar a Diferença entre Duas Distribuições de Probabilidade

A Cross-Entropy (ou entropia cruzada, se você preferir o termo em português) é fácil de implementar como parte de um programa que aprende usando gradiente descendente e backpropagation. Faremos isso nos próximos capítulos quando treinarmos uma rede completa, desenvolvendo uma versão melhorada do nosso programa anterior para classificar os dígitos manuscritos do dataset MNIST. O novo programa é chamado de network2.py e incorpora não apenas a entropia cruzada, mas também várias outras técnicas que estudaremos mais adiante. Agora, vejamos como usar a Entropia Cruzada Para Quantificar a Diferença Entre Duas Distribuições de Probabilidade.

Por enquanto, vamos ver como nosso novo programa classifica os dígitos MNIST. Usaremos uma rede com 30 neurônios ocultos, e usaremos um tamanho de mini-lote de 10. Definimos a taxa de aprendizado para $\eta = 0,5$ e nós treinamos por 30 épocas. A interface para o network2.py será um pouco diferente do network.py, mas ainda deve estar claro o que está acontecendo. Nos próximos capítulos apresentamos o código completo no repositório do livro no Github.

```
>>> import mnist_loader
>>> training_data, validation_data, test_data = \
... mnist_loader.load_data_wrapper()
>>> import network2
>>> net = network2.Network([784, 30, 10], cost=network2.CrossEntropyCost)
>>> net.large_weight_initializer()
>>> net.SGD(training_data, 30, 10, 0.5, evaluation_data=test_data,
... monitor_evaluation_accuracy=True)
```

Perceba que o comando net.large_weight_initializer() é usado para inicializar os pesos e vieses da mesma maneira que já descrevemos anteriormente. Precisamos executar este comando porque mais adiante vamos alterar o peso padrão para inicialização em nossas redes. O resultado da execução da sequência de comandos acima é uma rede com 95,49% de precisão.

Vejamos também o caso em que usamos 100 neurônios ocultos, a entropia cruzada, e mantemos os parâmetros da mesma forma. Neste caso, obtemos uma precisão de 96,82%. Essa é uma melhoria substancial em relação aos resultados que obtivemos nos capítulos anteriores, onde a precisão de classificação foi de 96,59%, usando o custo quadrático. Isso pode parecer uma pequena mudança, mas considere que a taxa de erro caiu de 3,41% para 3,18%. Ou seja, eliminamos cerca de um em quatorze dos erros originais. Isso é uma melhoria bastante útil.

É encorajador que o custo de entropia cruzada nos dê resultados semelhantes ou melhores do que o custo quadrático. No entanto, esses resultados não provam conclusivamente que a entropia cruzada é uma escolha melhor. A razão é que nós colocamos apenas um pequeno esforço na escolha de hyperparâmetros como taxa de aprendizado, tamanho de mini-lote e assim por diante. Para que a melhoria seja realmente convincente, precisaríamos fazer um

trabalho completo de otimização desses hyperparâmetros. Ainda assim, os resultados são encorajadores e reforçam nosso argumento teórico anterior de que a entropia cruzada é uma escolha melhor do que o custo quadrático.

Isso, a propósito, é parte de um padrão geral que veremos nos próximos capítulos e, na verdade, em grande parte do restante do livro. Vamos desenvolver uma nova técnica, vamos experimentá-la e obteremos resultados "aprimorados". É claro que é bom vermos essas melhorias, mas a interpretação de tais melhorias é sempre problemática. Elas só são verdadeiramente convincentes se virmos uma melhoria depois de nos esforçarmos para otimizar todos os outros hyperparâmetros. Isso é uma grande quantidade de trabalho, exigindo muito poder de computação, e normalmente não vamos fazer uma investigação tão exaustiva. Em vez disso, procederemos com base em testes informais como os realizados até aqui.

Até agora, discutimos a entropia cruzada de forma bem detalhada. Por que tanto esforço quando a entropia cruzada nos dá apenas uma pequena melhora em nossos resultados com o dataset MNIST? Mais adiante veremos outras técnicas, notadamente a regularização, que trazem melhorias muito maiores. Então, por que tanto foco na entropia cruzada? Parte da razão é que a entropia cruzada é uma função de custo amplamente utilizada e, portanto, vale a pena compreendê-la bem. Mas a razão mais importante é que a saturação dos neurônios é um problema importante nas redes neurais, um problema ao qual voltaremos repetidamente ao longo do livro. Por isso discutimos a entropia cruzada em extensão pois é um bom laboratório para começar a entender a saturação dos neurônios e como ela pode ser abordada.

O que significa a entropia cruzada? De onde isso vem?

Nossa discussão sobre a entropia cruzada se concentrou na análise algébrica e na implementação prática. Isso é útil, mas deixa questões conceituais mais amplas não respondidas, como: o que significa a entropia cruzada? Existe alguma maneira intuitiva de pensar sobre a entropia cruzada? E quanto ao significado intuitivo da entropia cruzada? Como devemos pensar sobre isso?

Explicar isso em profundidade nos levaria mais longe do que queremos ir neste livro. No entanto, vale ressaltar que existe uma maneira padrão de interpretar a entropia cruzada que vem do campo da teoria da informação. Vejamos.

Já sabemos que para treinar uma rede neural, você precisa encontrar o erro entre as saídas calculadas e as saídas alvo desejadas. A medida de erro mais comum é chamada de erro quadrático médio (ou Mean Square Error). No entanto, existem alguns resultados de pesquisa que sugerem o uso de uma medida diferente, denominada erro de entropia cruzada, como método preferível em relação ao erro quadrático médio.

A medida de entropia cruzada tem sido utilizada como alternativa ao erro quadrático médio. A entropia cruzada pode ser usada como uma medida de erro quando as saídas de uma rede podem ser pensadas como representando hipóteses independentes (por exemplo, cada nó significa um conceito diferente) e as ativações dos nós podem ser entendidas como representando a probabilidade (ou a confiança) que cada uma das hipóteses pode ser verdadeira. Nesse caso, o vetor de saída representa uma distribuição de probabilidade, e nossa medida de erro – entropia cruzada – indica a distância entre o que a rede acredita que essa distribuição deve ser e o que realmente deveria ser. Existe também uma razão prática para usar a entropia cruzada. Pode ser mais útil em problemas nos quais os alvos são 0 e 1. A entropia

cruzada tende a permitir que erros alterem pesos mesmo quando houver nós saturados (o que significa que suas derivadas são próximas de 0). Vamos compreender melhor isso:

A entropia cruzada é comumente usada para quantificar a diferença entre duas distribuições de probabilidade. Geralmente, a distribuição "verdadeira" (dos dados usados para treinamento) é expressa em termos de uma distribuição One-Hot.

Por exemplo, suponha que para uma instância de treinamento específica (uma única linha no seu dataset), a classe seja B (de 3 possíveis possibilidades: A, B e C). A distribuição única para esta instância de treinamento é, portanto:

Pr(Class A)	Pr(Class B)	Pr(Class C)
0.0	1.0	0.0

Você pode interpretar a distribuição acima da seguinte forma: a instância de treinamento tem 0% de probabilidade de ser classe A, 100% de probabilidade de ser classe B e 0% de probabilidade de ser a classe C.

Agora, suponha que seu algoritmo de aprendizado de máquina tenha previsto a seguinte distribuição de probabilidade:

Pr(Class A)	Pr(Class B)	Pr(Class C)
0.228	0.619	0.153

Quão próxima é a distribuição prevista da distribuição verdadeira? **É isso que determina o erro de entropia cruzada**. A entropia cruzada é representada por esta fórmula:

$$H(p, q) = -\sum_{x} p(x) \log q(x).$$

A soma é sobre as três classes A, B e C. Se você completar o cálculo, você achará que a perda é 0.479. Então, é assim que "longe" está a sua previsão da distribuição verdadeira.

A entropia cruzada é uma das muitas funções de perda possíveis. Essas funções de perda são tipicamente escritas como J(theta) e podem ser usadas dentro da descida do gradiente, que é uma estrutura iterativa para mover os parâmetros (ou coeficientes) para os valores ótimos.

A entropia cruzada descreve a perda entre duas distribuições de probabilidade.

Ao usar uma rede neural para realizar classificação e predição, geralmente é melhor usar o erro de entropia cruzada do que o erro de classificação e um pouco melhor usar o erro de entropia cruzada do que o erro quadrático médio para avaliar a qualidade da rede neural. É importante deixar claro que estamos lidando apenas com uma rede neural que é usada para classificar os dados, como a previsão da concessão de crédito (sim ou não), ou ainda outras classificações como idade, sexo ou dígitos no dataset MNIST e assim por diante. Não estamos lidando com uma rede neural que faz regressão, onde o valor a ser previsto é numérico.

Até o próximo capítulo!

Referências:

Formação Inteligência Artificial

Dot Product

Neural Networks & The Backpropagation Algorithm, Explained

Derivada

Machine Learning

The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition

Gradient Descent For Machine Learning

Pattern Recognition and Machine Learning

Understanding Activation Functions in Neural Networks

Redes Neurais, princípios e práticas

CAPÍTULO 19

Overfitting e Regularização – Parte 1

O físico Enrico Fermi, ganhador do Prêmio Nobel de Física em 1938, foi questionado sobre sua opinião em relação a um modelo matemático que alguns colegas haviam proposto como a solução para um importante problema de física não resolvido. O modelo teve excelente performance no experimento, mas Fermi estava cético. Ele perguntou quantos parâmetros livres poderiam ser definidos no modelo. "Quatro" foi a resposta. Fermi respondeu: "Eu lembro que meu amigo Johnny Von Neumann costumava dizer: com quatro parâmetros eu posso encaixar um elefante, e com cinco eu posso fazê-lo mexer seu tronco" *. Com isso, ele quis dizer que não se deve ficar impressionado quando um modelo complexo se ajusta bem a um conjunto de dados. Com parâmetros suficientes, você pode ajustar qualquer conjunto de dados.

(* A citação vem de um artigo de Freeman Dyson, que é uma das pessoas que propôs o modelo. O artigo "Um elefante de quatro parâmetros" ou "A four-parameter elephant" pode ser encontrado aqui.)

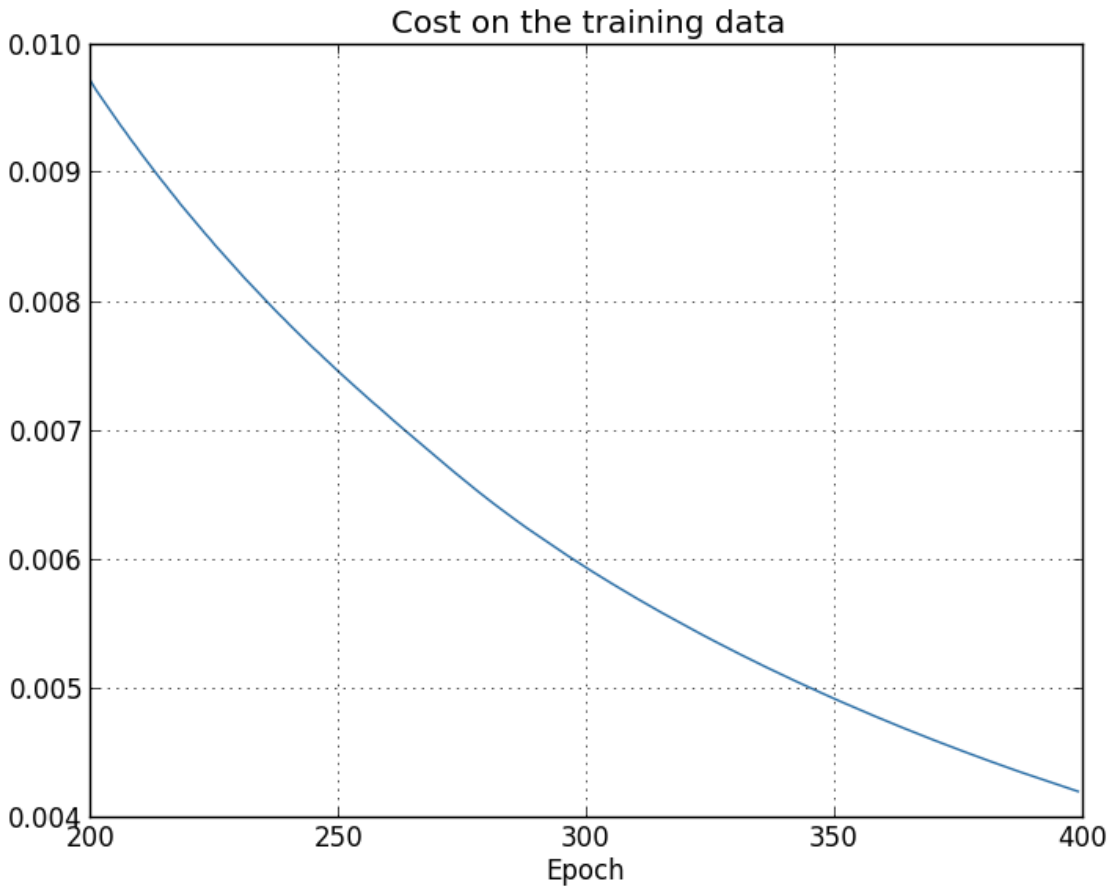
O ponto, claro, é que modelos com um grande número de parâmetros podem descrever uma variedade incrivelmente ampla de fenômenos. Mesmo que tal modelo esteja de acordo com os dados disponíveis, isso não o torna um bom modelo. Isso pode significar apenas que há liberdade suficiente no modelo que pode descrever quase qualquer conjunto de dados de tamanho determinado, sem capturar nenhuma percepção genuína do fenômeno em questão. Quando isso acontece, o modelo funcionará bem para os dados existentes, mas não conseguirá generalizar para novas situações. O verdadeiro teste de um modelo é sua capacidade de fazer previsões em situações que não foram expostas antes.

Fermi e von Neumann suspeitavam de modelos com quatro parâmetros. Nossa rede de 30 neurônios ocultos para classificação de dígitos MNIST possui quase 24.000 parâmetros! Nossa rede de 100 neurônios ocultos tem cerca de 80.000 parâmetros e redes neurais profundas de última geração às vezes contêm milhões ou até bilhões de parâmetros. Devemos confiar nos resultados?

Vamos aguçar este problema construindo uma situação em que a nossa rede faz um mau trabalho ao generalizar para novas situações. Usaremos nossa rede de 30 neurônios ocultos, com seus 23.860 parâmetros. Mas não treinamos a rede usando todas as imagens de treinamento de 50.000 dígitos MNIST. Em vez disso, usaremos apenas as primeiras 1.000 imagens de treinamento. Usar esse conjunto restrito tornará o problema com a generalização muito mais evidente. Vamos treinar usando a função de custo de entropia cruzada, com uma taxa de aprendizado de $\eta = 0,5$ e um tamanho de mini-lote de 10. No entanto, vamos treinar por 400 épocas, pois não estamos usando muitos exemplos de treinamento. Vamos usar network2 para ver como a função de custo muda (o código você encontra no repositório do curso no Github):

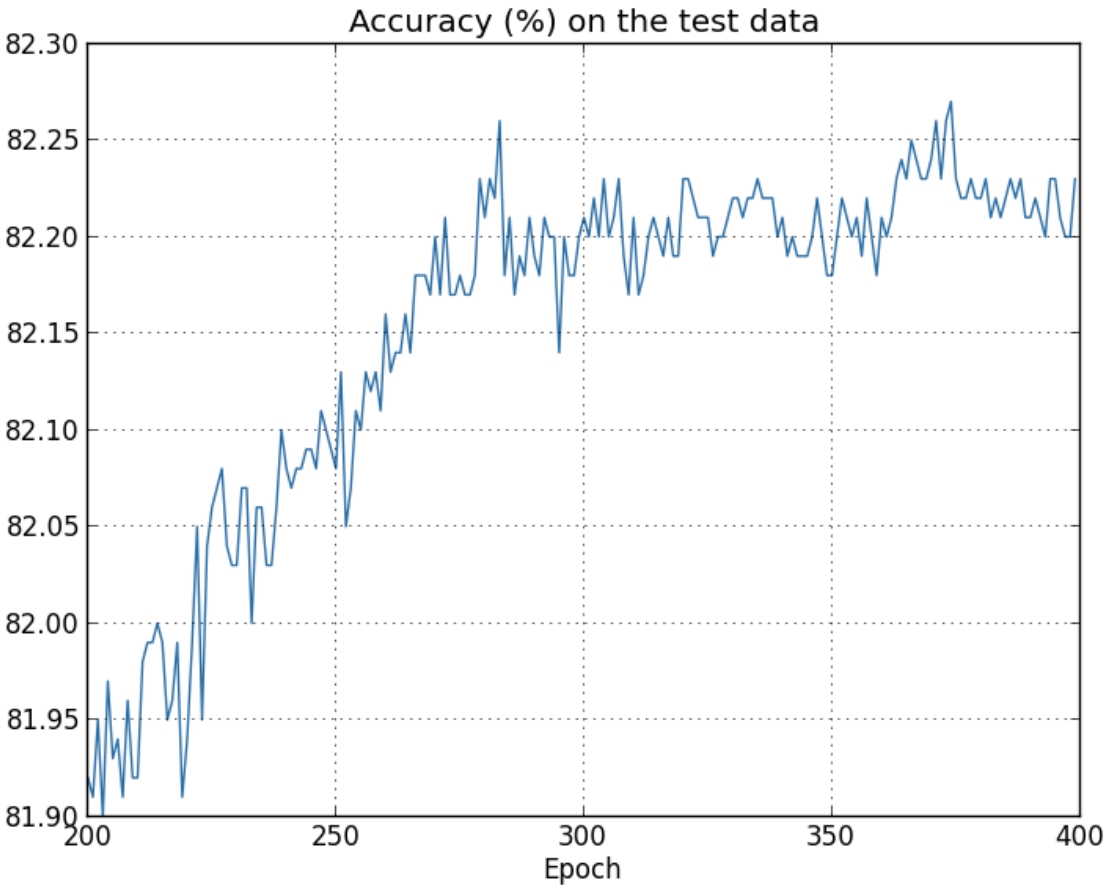
```
>>> import mnist_loader
>>> training_data, validation_data, test_data = \
... mnist_loader.load_data_wrapper()
>>> import network2
>>> net = network2.Network([784, 30, 10], cost=network2.CrossEntropyCost)
>>> net.large_weight_initializer()
>>> net.SGD(training_data[:1000], 400, 10, 0.5, evaluation_data=test_data,
... monitor_evaluation_accuracy=True, monitor_training_cost=True)
```

Usando os resultados, podemos traçar a maneira como o custo muda à medida que a rede aprende (o script overfitting.py contém o código que gera esse resultado):



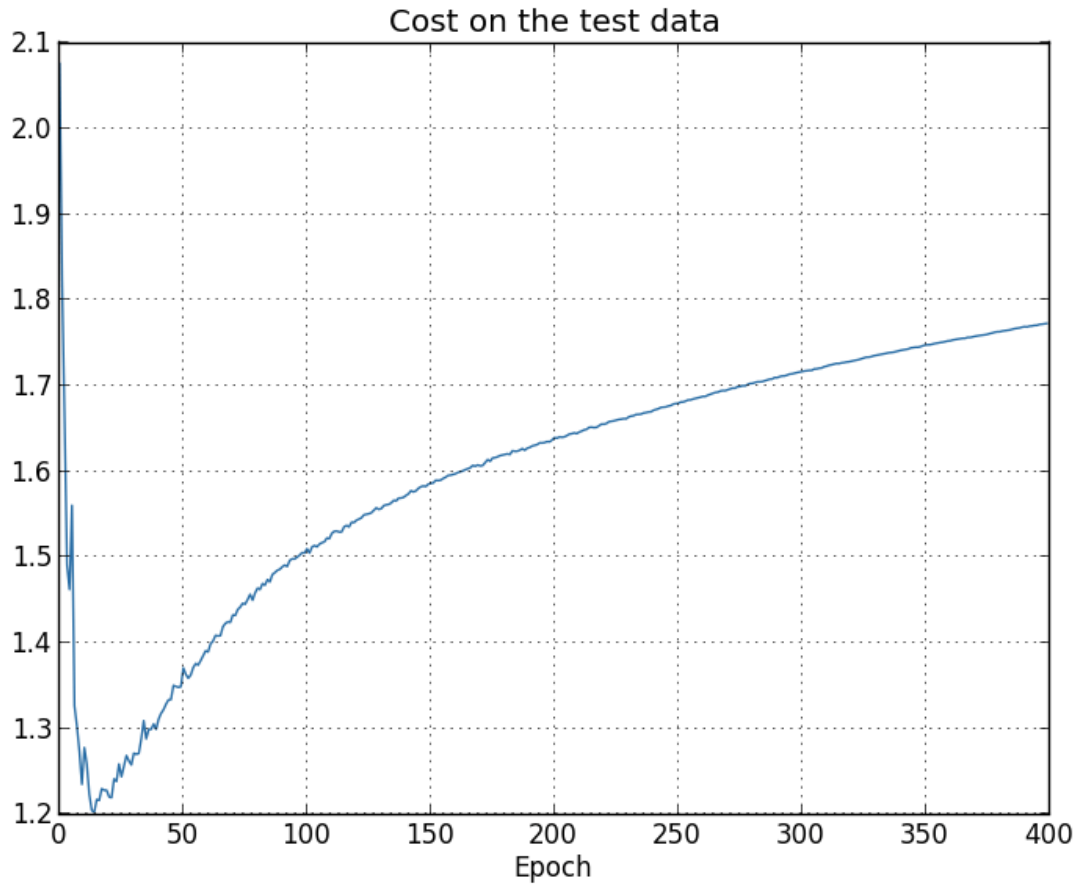
Isso parece encorajador, mostrando uma redução suave no custo, exatamente como esperamos. Note que eu só mostrei as épocas de treinamento de 200 a 399. Isso nos dá uma boa visão dos últimos estágios do aprendizado, que, como veremos, é onde está a ação interessante.

Vamos agora ver como a precisão da classificação nos dados de teste muda com o tempo:



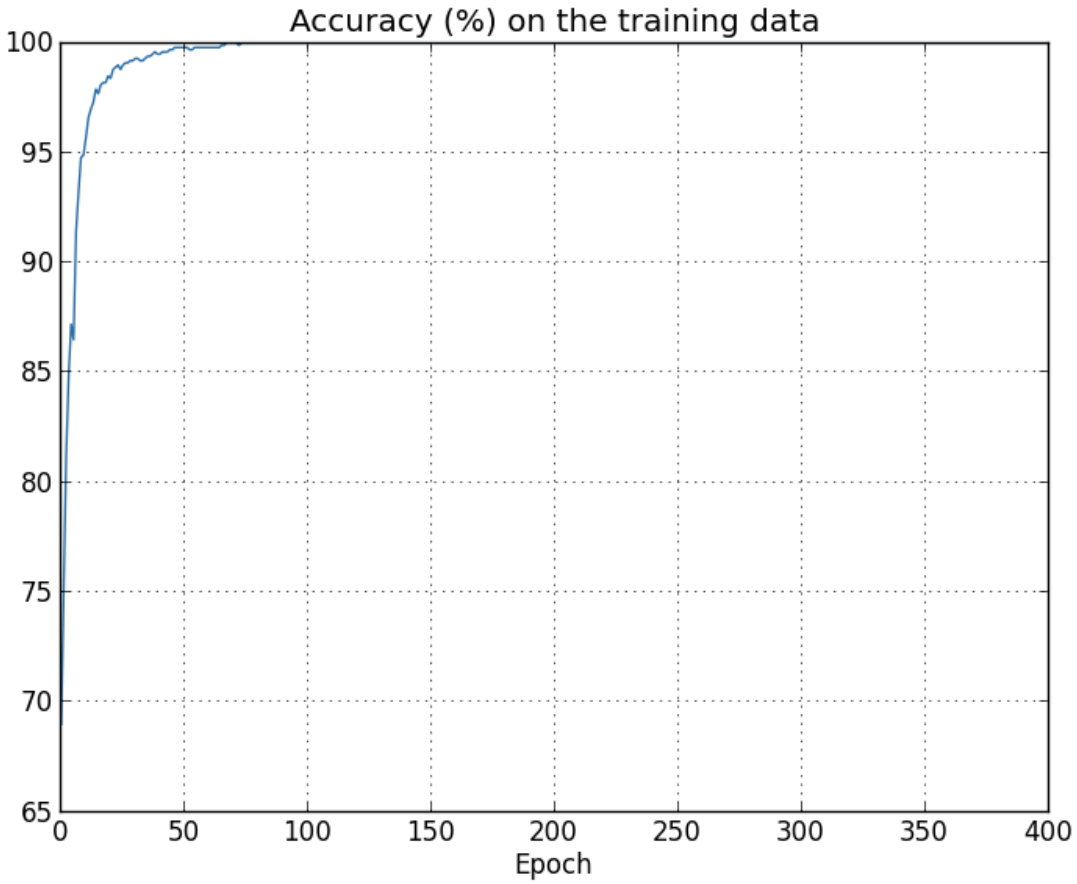
Mais uma vez, eu ampliei um pouco. Nas primeiras 200 épocas (não mostradas), a precisão sobe para pouco menos de 82%. O aprendizado então diminui gradualmente. Finalmente, por volta da época 280, a precisão da classificação praticamente pára de melhorar. As épocas posteriores meramente vêem pequenas flutuações estocásticas perto do valor da precisão na época 280. Compare isso com o gráfico anterior, em que o custo associado aos dados de treinamento continua a cair suavemente. Se olharmos apenas para esse custo, parece que nosso modelo ainda está ficando "melhor". Mas os resultados da precisão do teste mostram que a melhoria é uma ilusão. Assim como o modelo que Fermi não gostava, o que nossa rede aprende após a época 280 não mais se generaliza para os dados de teste. E assim não é um aprendizado útil. Dizemos que a rede está super adaptando ou com sobreajuste ou ainda com overfitting, a partir da época 280.

Você pode se perguntar se o problema aqui é que eu estou olhando para o custo dos dados de treinamento, ao contrário da precisão da classificação nos dados de teste. Em outras palavras, talvez o problema seja que estamos fazendo uma comparação de maçãs e laranjas. O que aconteceria se comparássemos o custo dos dados de treinamento com o custo dos dados de teste, estaríamos comparando medidas semelhantes? Ou talvez pudéssemos comparar a precisão da classificação tanto nos dados de treinamento quanto nos dados de teste? Na verdade, essencialmente o mesmo fenômeno aparece, não importa como fazemos a comparação. Os detalhes mudam, no entanto. Por exemplo, vamos analisar o custo nos dados de teste:



Podemos ver que o custo nos dados de teste melhora até a época 15, mas depois disso ele realmente começa a piorar, mesmo que o custo nos dados de treinamento continue melhorando. Este é outro sinal de que nosso modelo está super adaptando (overfitting). No entanto, coloca um enigma, que é se devemos considerar a época 15 ou a época 280 como o ponto em que o overfitting está dominando a aprendizagem? Do ponto de vista prático, o que realmente nos importa é melhorar a precisão da classificação nos dados de teste, enquanto o custo dos dados de teste não é mais do que um proxy para a precisão da classificação. E assim faz mais sentido considerar a época 280 como o ponto além do qual o overfitting está dominando o aprendizado em nossa rede neural.

Outro sinal de overfitting pode ser visto na precisão da classificação nos dados de treinamento:



A precisão aumenta até 100%. Ou seja, nossa rede classifica corretamente todas as 1.000 imagens de treinamento! Enquanto isso, nossa precisão de teste atinge apenas 82,27%. Portanto, nossa rede realmente está aprendendo sobre as peculiaridades do conjunto de treinamento, não apenas reconhecendo os dígitos em geral. É quase como se nossa rede estivesse apenas memorizando o conjunto de treinamento, sem entender os dígitos suficientemente bem para generalizar o conjunto de testes.

Overfitting é um grande problema em redes neurais. Isso é especialmente verdadeiro em redes modernas, que geralmente têm um grande número de pesos e vieses. Para treinar de forma eficaz, precisamos de uma maneira de detectar quando o overfitting está acontecendo. E precisamos aplicar técnicas para reduzir os efeitos do overfitting (por todo esse trabalho e conhecimento necessário, Cientistas de Dados devem ser muito bem remunerados).

A maneira óbvia de detectar overfitting é usar a abordagem acima, mantendo o controle da precisão nos dados de teste conforme nossos treinos da rede. Se percebermos que a precisão nos dados de teste não está mais melhorando, devemos parar de treinar. É claro que, estritamente falando, isso não é necessariamente um sinal de overfitting. Pode ser que a precisão nos dados de teste e os dados de treinamento parem de melhorar ao mesmo tempo. Ainda assim, a adoção dessa estratégia impedirá o overfitting.

Na verdade, usaremos uma variação dessa estratégia. Lembre-se de que, quando carregamos os dados MNIST, carregamos em três conjuntos de dados:

```
>>> import mnist_loader
>>> training_data, validation_data, test_data = \
... mnist_loader.load_data_wrapper()
```

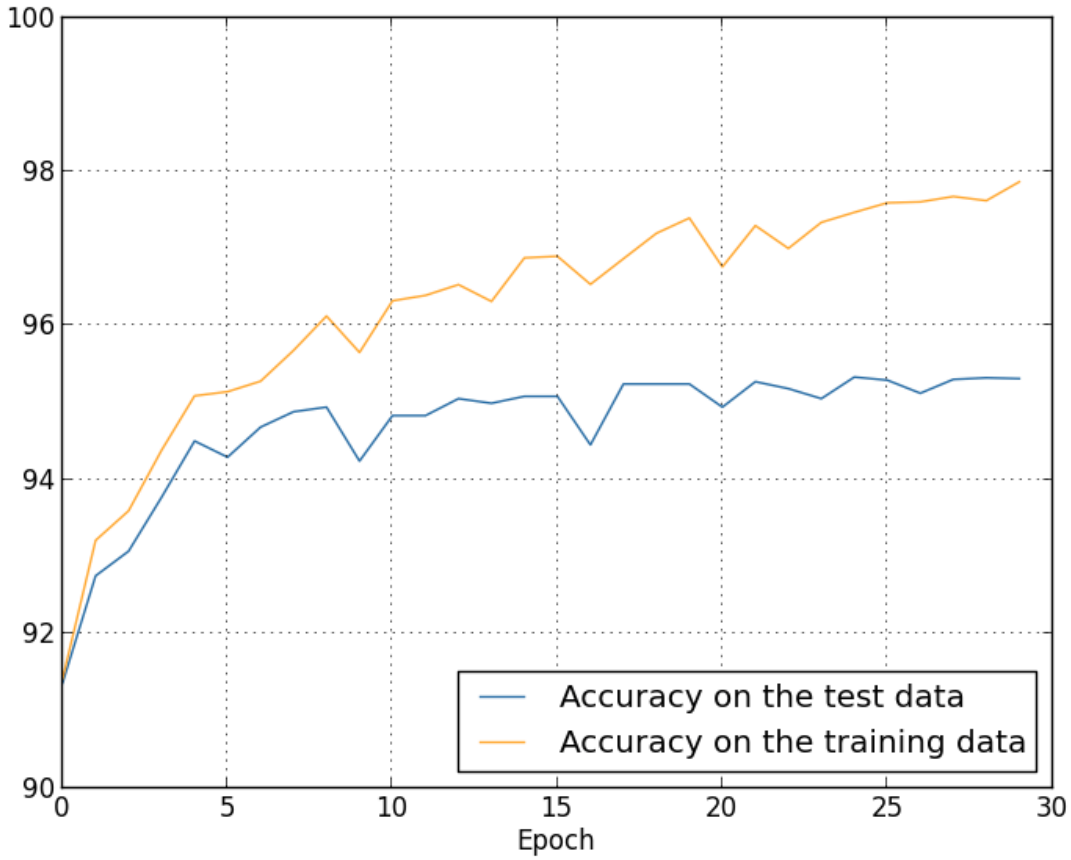
Até agora, usamos o **training_data** e **test_data** e ignoramos o **validation_data**. O validation_data contém 10.000 imagens de dígitos, imagens que são diferentes das 50.000 imagens no conjunto de treinamento MNIST e das 10.000 imagens no conjunto de teste MNIST. Em vez de usar o test_data para evitar overfitting, usaremos o validation_data. Para fazer isso, usaremos praticamente a mesma estratégia descrita acima para o test_data. Ou seja, calcularemos a precisão da classificação nos dados de validação no final de cada época. Quando a precisão da classificação nos dados de validação estiver saturada, paramos de treinar. Essa estratégia é chamada de parada antecipada (Early-Stopping). É claro que, na prática, não sabemos imediatamente quando a precisão está saturada. Em vez disso, continuamos treinando até termos certeza de que a precisão está saturada.

Por que usar o validation_data para evitar overfitting, em vez de test_data? Na verdade, isso faz parte de uma estratégia mais geral, que é usar o validation_data para avaliar diferentes opções de avaliação de hiperparâmetros, como o número de épocas para treinamento, a taxa de aprendizado, a melhor arquitetura de rede e assim por diante. Usamos essas avaliações para encontrar e definir bons valores para os hiperparâmetros. De fato, embora eu não tenha mencionado isso até agora, isto é, em parte, como chegamos às escolhas de hiperparâmetros feitas anteriormente neste livro. (Mais sobre isso depois.)

Claro, isso não responde de forma alguma à pergunta de por que estamos usando o validation_data para evitar overfitting, em vez de test_data. Para entender o porquê, considere que, ao definir os hiperparâmetros, é provável que tentemos muitas opções diferentes para os hiperparâmetros. Se definirmos os hiperparâmetros com base nas avaliações do test_data, será possível acabarmos super adequando nossos hiperparâmetros ao test_data. Ou seja, podemos acabar encontrando hiperparâmetros que se encaixam em peculiaridades particulares dos dados de teste, mas onde o desempenho da rede não se generalizará para outros conjuntos de dados. Protegemos contra isso descobrindo os hiperparâmetros usando o validation_data. Então, uma vez que tenhamos os hiperparâmetros que queremos, fazemos uma avaliação final da precisão usando o test_data. Isso nos dá confiança de que nossos resultados nos dados de teste são uma medida real de quão bem nossa rede neural se generaliza. Para colocar de outra forma, você pode pensar nos dados de validação como um tipo de dados de treinamento que nos ajuda a aprender bons parâmetros. Essa abordagem para encontrar bons hiperparâmetros é às vezes conhecida como o método "hold out", uma vez que os dados de validação são mantidos separados ou "mantidos" a partir dos dados de treinamento.

Agora, na prática, mesmo depois de avaliar o desempenho nos dados de teste, podemos mudar nossa opinião e tentar outra abordagem – talvez uma arquitetura de rede diferente – que envolva a descoberta de um novo conjunto de hiperparâmetros. Se fizermos isso, não há perigo de acabarmos com o test_data também? Precisamos de uma regressão potencialmente infinita de conjuntos de dados, para que possamos ter certeza de que nossos resultados serão generalizados? Abordar essa preocupação é um problema profundo e difícil. Mas para nossos objetivos práticos, não vamos nos preocupar muito com essa questão. Em vez disso, vamos nos concentrar no método básico de retenção, com base nos dados training_data, validation_data e test_data, conforme descrito acima.

Vimos que o overfitting ocorre quando estamos usando apenas 1.000 imagens de treinamento. O que acontece quando usamos o conjunto completo de treinamento de 50.000 imagens? Manteremos todos os outros parâmetros iguais (30 neurônios ocultos, taxa de aprendizado de 0,5, tamanho de mini-lote de 10), mas treinamos usando todas as 50.000 imagens por 30 épocas. Aqui está um gráfico mostrando os resultados da precisão de classificação nos dados de treinamento e nos dados de teste. Observe que usei os dados de teste aqui, em vez dos dados de validação, para tornar os resultados mais diretamente comparáveis aos gráficos anteriores.



Como você pode ver, a precisão nos dados de teste e treinamento permanece muito mais próxima do que quando estávamos usando 1.000 exemplos de treinamento. Em particular, a melhor precisão de classificação de 97,86% nos dados de treinamento é apenas 2,53% maior do que os 95,33% nos dados de teste. Isso é comparado com a diferença de 17,73% que tivemos anteriormente! Overfitting ainda está acontecendo, mas foi bastante reduzido. Nossa rede está se generalizando muito melhor dos dados de treinamento para os dados de teste. Em geral, uma das melhores maneiras de reduzir o overfitting é aumentar o volume (tamanho) dos dados de treinamento (fica claro agora porque Big Data está revolucionando a Ciência de Dados?). Com dados de treinamento suficientes, é difícil até mesmo uma rede muito grande sofrer de overfitting. Infelizmente, os dados de treinamento podem ser caros ou difíceis de adquirir, por isso nem sempre é uma opção prática.

Aumentar a quantidade de dados de treinamento é uma maneira de reduzir o overfitting. Mas existem outras maneiras de reduzir a extensão do overfitting? Uma abordagem possível é reduzir o tamanho da nossa rede. No entanto, redes grandes têm o potencial de serem mais poderosas do que redes pequenas e, portanto, essa é uma opção que só adotamos em último caso.

Felizmente, existem outras técnicas que podem reduzir o overfitting, mesmo quando temos uma rede fixa e dados de treinamento fixos. Estas técnicas são conhecidas como técnicas de regularização e serão assunto do próximo capítulo.

Até lá!

Referências:

Formação Inteligência Artificial

Dot Product

Neural Networks & The Backpropagation Algorithm, Explained

Neural Networks and Deep Learning

Machine Learning

The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition

Gradient Descent For Machine Learning

Pattern Recognition and Machine Learning

Understanding Activation Functions in Neural Networks

Redes Neurais, princípios e práticas

CAPÍTULO 20

Overfitting e Regularização – Parte 2

Aumentar a quantidade de dados de treinamento é uma maneira de reduzir o overfitting. Mas existem outras maneiras de reduzir a extensão de ocorrência do overfitting? Uma abordagem possível é reduzir o tamanho da nossa rede. No entanto, redes grandes têm o potencial de serem mais poderosas do que redes pequenas e essa é uma opção que só adotaríamos com relutância.

Felizmente, existem outras técnicas que podem reduzir o overfitting, mesmo quando temos uma rede de tamanho fixo e dados de treinamento em quantidade limitada. Essas técnicas são conhecidos como **técnicas de regularização**. Neste capítulo descrevemos uma das técnicas de regularização mais comumente usadas, uma técnica às vezes conhecida como decaimento de peso (weight decay) ou Regularização L2. A ideia da Regularização L2 é adicionar um termo extra à função de custo, um termo chamado termo de regularização. Aqui está a entropia cruzada regularizada:

$$C = -\frac{1}{n} \sum_{xj} \left[y_j \ln a_j^L + (1 - y_j) \ln(1 - a_j^L) \right] + \frac{\lambda}{2n} \sum_w w^2$$

Equação 1

O primeiro termo é apenas a expressão usual para a entropia cruzada. Mas adicionamos um segundo termo, a soma dos quadrados de todos os pesos da rede. Isto é escalonado por um fator λ / 2n, onde $\lambda > 0$ é conhecido como o parâmetro de regularização e **n** é, como de costume, o tamanho do nosso conjunto de treinamento. Vou discutir mais tarde como λ é escolhido. É importante notar também que o termo de regularização não inclui os vieses. Eu também voltarei a isso mais frente.

Claro, é possível regularizar outras funções de custo, como o custo quadrático. Isso pode ser feito de maneira semelhante:

$$C = \frac{1}{2n} \sum_x \|y - a^L\|^2 + \frac{\lambda}{2n} \sum_w w^2$$

Equação 2

Em ambos os casos, podemos escrever a função de custo regularizada como:

$$C = C_0 + \frac{\lambda}{2n} \sum_w w^2$$

Equação 3

onde C0 é a função de custo original e não regularizada.

Intuitivamente, o efeito da regularização é fazer com que a rede prefira aprender pequenos pesos, sendo todas as outras coisas iguais. Pesos grandes só serão permitidos se melhorarem consideravelmente a primeira parte da função de custo. Dito de outra forma, a regularização pode ser vista como uma forma de se comprometer entre encontrar pequenos pesos e minimizar a função de custo original. A importância relativa dos dois elementos do compromisso depende do valor de λ: quando λ é pequeno, preferimos minimizar a função de custo original, mas quando λ é grande, preferimos pesos pequenos.

Agora, não é de todo óbvio porque fazer este tipo de compromisso deve ajudar a reduzir o overfitting! Mas acontece que sim, reduz. Abordaremos a questão de porque isso ajuda na redução do overfitting no próximo capítulo, mas primeiro vamos trabalhar em um exemplo mostrando como a regularização reduz o overfitting.

Para construir um exemplo, primeiro precisamos descobrir como aplicar nosso algoritmo de aprendizado de descida de gradiente estocástico em uma rede neural regularizada. Em particular, precisamos saber como calcular as derivadas parciais ∂C/∂w e ∂C/∂b para todos os pesos e vieses na rede. Tomando as derivadas parciais da Equação 3 acima, temos:

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} w$$
$$\frac{\partial C}{\partial b} = \frac{\partial C_0}{\partial b}.$$

Equação 4

Os termos $\partial C0/\partial w$ e $\partial C0/\partial b$ podem ser calculados usando backpropagation, conforme descrito nos capítulos anteriores. E assim vemos que é fácil calcular o gradiente da função de custo regularizada, pois basta usar backpropagation, como de costume, e depois adicionar **(λ/n).w** à derivada parcial de todos os termos de peso. As derivadas parciais em relação aos vieses são inalteradas e, portanto, a regra de aprendizado de descida de gradiente para os vieses não muda da regra usual:

$$b \rightarrow b - \eta \frac{\partial C_0}{\partial b}$$

Equação 5

A regra de aprendizado para os pesos se torna:

$$\begin{aligned} w &\rightarrow w - \eta \frac{\partial C_0}{\partial w} - \frac{\eta \lambda}{n} w \\ &= \left(1 - \frac{\eta \lambda}{n}\right) w - \eta \frac{\partial C_0}{\partial w} \end{aligned}$$

Equação 6

Isto é exatamente o mesmo que a regra usual de aprendizado de descida de gradiente, exceto pelo fato de primeiro redimensionarmos o peso w por um fator $1 - (\eta\lambda/n)$. Esse *reescalonamento* é, às vezes, chamado de redução de peso, uma vez que diminui os pesos. À primeira vista, parece que isso significa que os pesos estão sendo direcionados para zero, mas isso não é bem isso, uma vez que o outro termo pode levar os pesos a aumentar, se isso causar uma diminuição na função de custo não regularizada.

Ok, é assim que a descida de gradiente funciona. E quanto à descida de gradiente estocástica? Bem, assim como na descida de gradiente estocástica não-regularizada, podemos estimar

$\partial C0/\partial w$ pela média de um mini-lote de m exemplos de treinamento. Assim, a regra de aprendizagem regularizada para a descida de gradiente estocástica torna-se:

$$w \to \left(1 - \frac{\eta\lambda}{n}\right) w - \frac{\eta}{m} \sum_x \frac{\partial C_x}{\partial w}$$

Equação 7

onde a soma é sobre exemplos de treinamento x no mini-lote, e Cx é o custo (não-regularizado) para cada exemplo de treinamento. Isto é exatamente o mesmo que a regra usual para descida de gradiente estocástico, exceto pelo fator de decaimento de peso de $1 - (\eta\lambda/n)$. Finalmente, e por completo, deixe-me declarar a regra de aprendizagem regularizada para os vieses. Isto é, naturalmente, exatamente o mesmo que no caso não regularizado:

$$b \to b - \frac{\eta}{m} \sum_x \frac{\partial C_x}{\partial b}$$

Equação 8

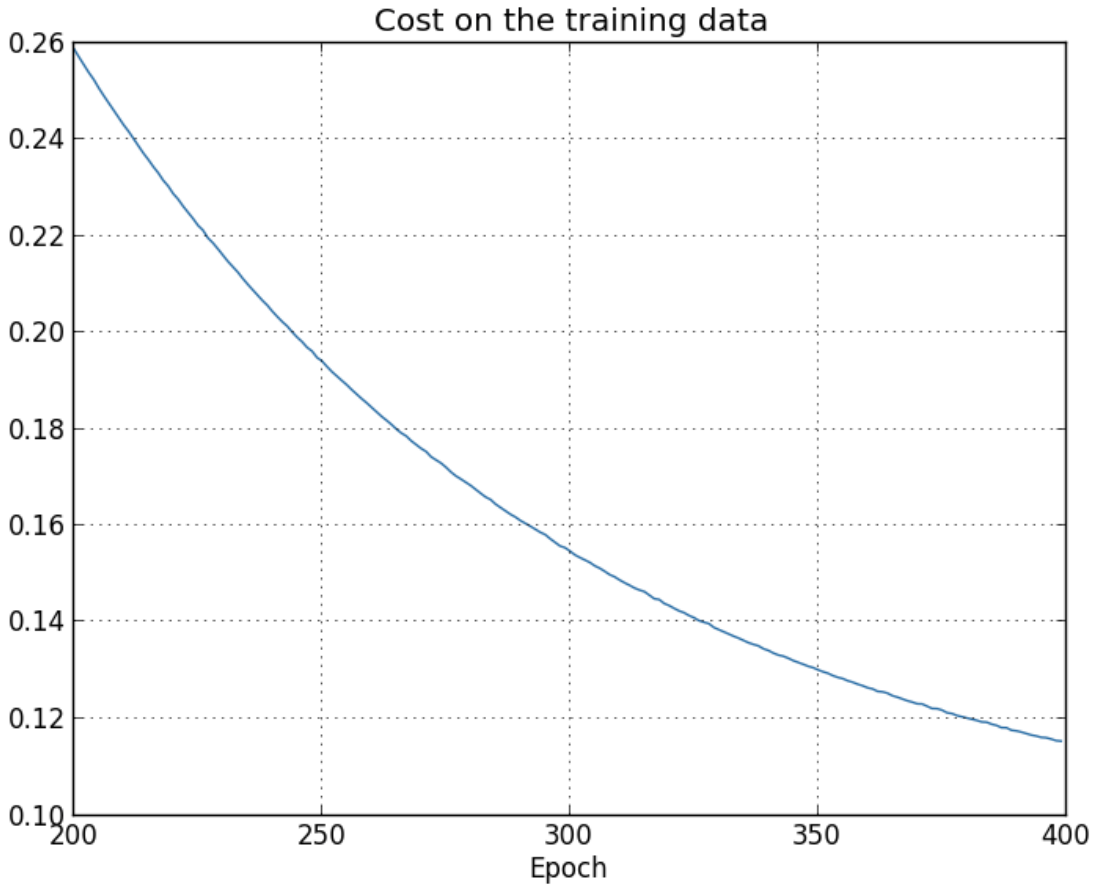
onde a soma é sobre exemplos de treinamento x no mini-lote.

Vamos ver como a regularização altera o desempenho da nossa rede neural. Usaremos uma rede com 30 neurônios ocultos, um tamanho de mini-lote de 10, uma taxa de aprendizado de 0,5 e a função de custo de entropia cruzada. No entanto, desta vez vamos usar um parâmetro de regularização de $\lambda = 0{,}1$. Note que no código, usamos o nome da variável **lmbda**, porque **lambda** é uma palavra reservada em Python, com um significado não relacionado ao que estamos fazendo aqui (caso tenha dúvidas sobre as palavras reservadas em Python, acesse o curso gratuito Python Fundamentos Para Análise de Dados – Capítulo 2).

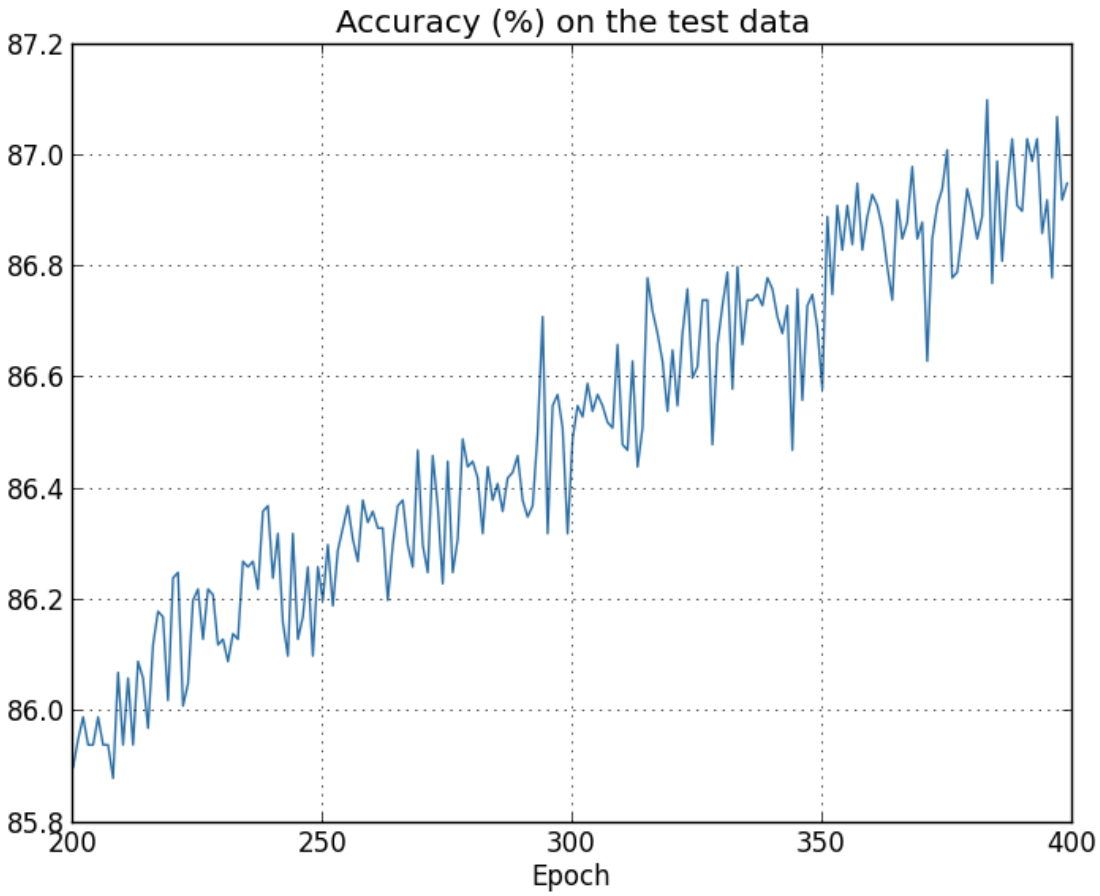
Eu também usei o test_data novamente, não o validation_data. Estritamente falando, devemos usar o validation_data, por todas as razões que discutimos anteriormente. Mas decidi usar o test_data porque ele torna os resultados mais diretamente comparáveis com nossos resultados anteriores e não regularizados. Você pode facilmente alterar o código para usar o validation_data e você verá que ele terá resultados semelhantes.

```
>>> import mnist_loader
>>> training_data, validation_data, test_data = \
... mnist_loader.load_data_wrapper()
>>> import network2
>>> net = network2.Network([784, 30, 10], cost=network2.CrossEntropyCost)
>>> net.large_weight_initializer()
>>> net.SGD(training_data[:1000], 400, 10, 0.5,
... evaluation_data=test_data, lmbda = 0.1,
... monitor_evaluation_cost=True, monitor_evaluation_accuracy=True,
... monitor_training_cost=True, monitor_training_accuracy=True)
```

O custo com os dados de treinamento diminui durante todo o tempo, da mesma forma que no caso anterior, não regularizado no capítulo anterior:



Mas desta vez a precisão no test_data continua a aumentar durante as 400 épocas:



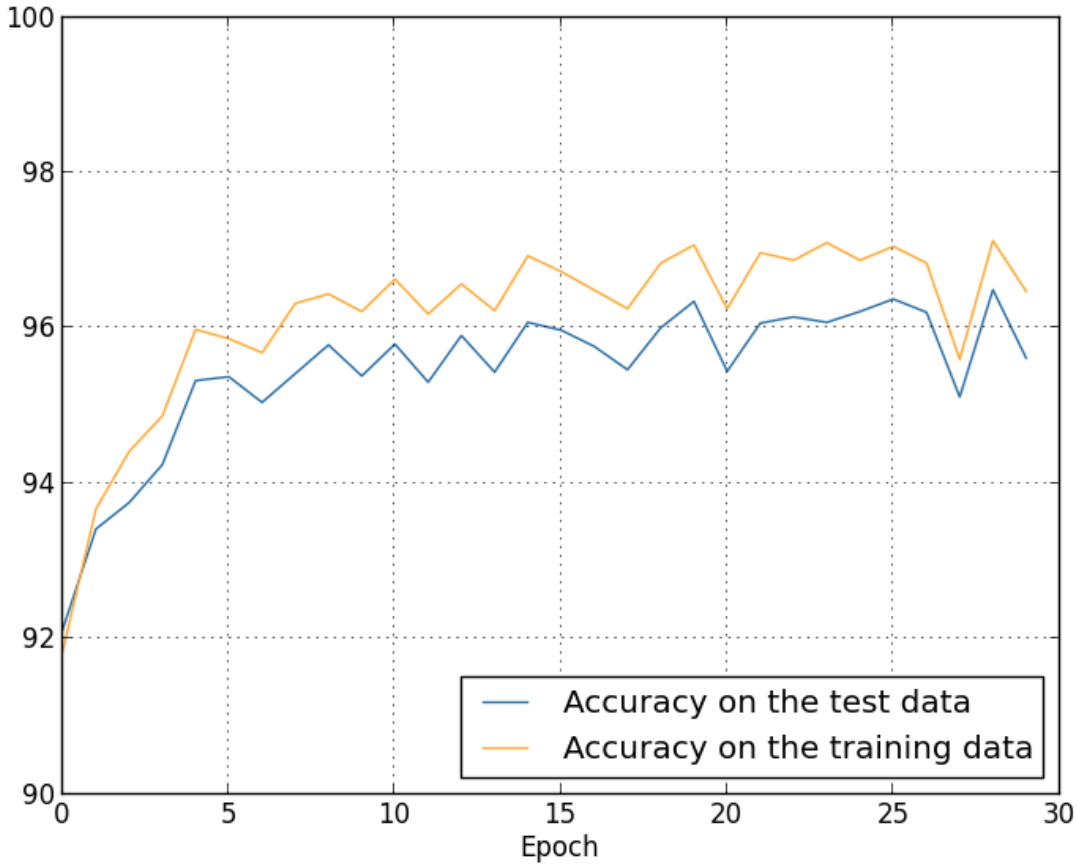
Claramente, o uso da regularização suprimiu o overfitting. Além do mais, a precisão é consideravelmente maior, com uma precisão de classificação de pico de 87.1%, em comparação com o pico de 82.27% obtido no caso não regularizado. De fato, quase certamente poderíamos obter resultados consideravelmente melhores, continuando a treinar mais de 400 épocas. Parece que, empiricamente, a regularização está fazendo com que nossa rede generalize melhor e reduza consideravelmente os efeitos do overfitting.

O que acontece se sairmos do ambiente artificial de ter apenas 1.000 imagens de treinamento e retornar ao conjunto completo de treinamento de 50.000 imagens? É claro, já vimos que o overfitting é muito menos problemático com as 50.000 imagens. A regularização ajuda ainda mais? Vamos manter os hiperparâmetros iguais ao exemplo anterior – 30 épocas, taxa de aprendizado de 0,5, tamanho de mini-lote de 10. No entanto, precisamos modificar o parâmetro de regularização. A razão é porque o tamanho n do conjunto de treinamento mudou de $n = 1.000$ para $n = 50.000$, e isso muda o fator de decaimento de peso $1 - (\eta\lambda/n)$. Se continuássemos a usar $\lambda = 0{,}1$, isso significaria muito menos perda de peso e, portanto, muito menos efeito de regularização. Nós compensamos mudando para $\lambda = 5.0$.

Ok, vamos treinar nossa rede, parando primeiro para reinicializar os pesos:

```
>>> net.large_weight_initializer()
>>> net.SGD(training_data, 30, 10, 0.5,
... evaluation_data=test_data, lmbda = 5.0,
... monitor_evaluation_accuracy=True, monitor_training_accuracy=True)
```

Obtemos os resultados:



Há muitas boas notícias aqui. Primeiro, nossa precisão de classificação nos dados de teste aumentou de 95.49%, quando não foi regularizada, para 96.49%. Isso é uma grande melhoria. Em segundo lugar, podemos ver que a diferença entre os resultados nos dados de treinamento e teste é muito menor do que antes, com um percentual abaixo de zero. Essa ainda é uma lacuna significativa, mas obviamente fizemos um progresso substancial para reduzir o overfitting.

Finalmente, vamos ver qual a precisão da classificação de teste que obtemos quando usamos 100 neurônios ocultos e um parâmetro de regularização de $\lambda = 5.0$. Eu não vou passar por uma análise detalhada de overfitting aqui, isso é puramente por diversão, só para ver a precisão que podemos obter quando usamos nossos novos truques: a função de custo de entropia cruzada e a Regularização L2.

```
>>> net = network2.Network([784, 100, 10], cost=network2.CrossEntropyCost)
>>> net.large_weight_initializer()
>>> net.SGD(training_data, 30, 10, 0.5, lmbda=5.0,
... evaluation_data=validation_data,
... monitor_evaluation_accuracy=True)
```

O resultado final é uma precisão de classificação de 97.92% nos dados de validação. É um grande salto do caso dos 30 neurônios ocultos. Na verdade, ajustando um pouco mais, para executar por 60 épocas com $\eta = 0.1$ e $\lambda = 5.0$, quebramos a barreira de 98%, alcançando uma precisão de classificação de 98.04% nos dados de validação. Nada mal para o que acaba sendo 152 linhas de código!

Descrevi a regularização como uma forma de reduzir o overfitting e aumentar as precisões de classificação. Na verdade, esse não é o único benefício. Empiricamente, ao executar várias execuções de nossas redes com o dataset MNIST, mas com diferentes inicializações de peso (aleatórias), descobrimos que as execuções não-regularizadas ocasionalmente ficarão "presas", aparentemente capturadas em mínimos locais da função de custo. O resultado é que diferentes execuções às vezes fornecem resultados bastante diferentes. Por outro lado, as execuções regularizadas forneceram resultados muito mais facilmente replicáveis.

Por que isso está acontecendo? Heuristicamente, se a função de custo for *desregularizada*, o comprimento do vetor de peso provavelmente crescerá, todas as outras coisas sendo iguais. Com o tempo, isso pode levar o vetor de peso a ser realmente muito grande. Isso pode fazer com que o vetor de peso fique preso apontando mais ou menos na mesma direção, já que as mudanças devido a descida do gradiente fazem apenas pequenas alterações na direção, quando o comprimento é longo. Acredito que esse fenômeno esteja dificultando o nosso algoritmo de aprendizado para explorar adequadamente o espaço de pesos e, consequentemente, mais difícil encontrar bons mínimos da função de custo.

Ainda não acabamos sobre regularização. Mais sobre isso no próximo capítulo! Até lá!

Referências:

Formação Inteligência Artificial

Dot Product

Neural Networks & The Backpropagation Algorithm, Explained

Neural Networks and Deep Learning

Machine Learning

The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition

Gradient Descent For Machine Learning

Pattern Recognition and Machine Learning

Understanding Activation Functions in Neural Networks

Redes Neurais, princípios e práticas